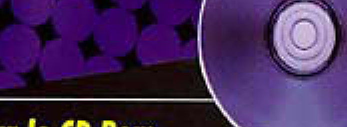
Anne Tasso

Le livre de Java premier langage

Pour les vrais débutants en programmation



Sur le CD-Rom offert avec ce livre :

- Java 2 SDK 1.3 : environnement de programmation complet pour réaliser et exécuter vos programmes Java sous Windows 95, 98, 2000 ou NT 4.
- Corrigé du projet et des exercices
- Code source de tous les programmes



Eyrolles


Le livre de
Java
premier langage

CHEZ LE MÊME ÉDITEUR

DANS LA MÊME COLLECTION

C. DELANNOY. – **Le livre du C premier langage**.
Pour les débutants en programmation. N°8838, 1994, 280 pages.

P. CHALÉAT, D. CHARNAY. – **Programmation HTML et JavaScript**.
N°9182, 1998, 460 pages.

C. DELANNOY. – **Programmer en langage C**. *Avec exercices corrigés.*
N°8985, 1996, 296 pages.

C. DELANNOY. – **La référence du C norme ANSI/ISO**.
N°9036, 1998, 950 pages.

C. DELANNOY. – **Programmer en langage C++**.
N°9138, 5e édition, 2000, 648 pages.

C. DELANNOY. – **Apprendre le C++ avec Visual C++ 6**.
N°9088, 1999, 496 pages.

C. DELANNOY. – **Exercices en langage C++**.
N°9067, 2e édition, 1999, 296 pages.

C. DELANNOY. – **Programmer en Turbo Pascal 7.0**.
N°8986, 1993-1997, 368 pages.

PROGRAMMATION INTERNET

A. PATZER *et al*. – **Programmation Java côté serveur**.
Servlets, JSP et EJB. N°9109, 2000, 984 pages.

P. CHAN. – **Le dictionnaire officiel Java 2**.
N°9089, 1999, 890 pages.

N. MCFARLANE. – **JavaScript professionnel**.
N°9141, 2000, 950 pages.

J.-C. BERNADAC, F. KNAB. – **Construire une application XML**.
N°9081, 1999, 500 pages.

A. HOMER, D. SUSSMAN, B. FRANCIS. – **ASP 3.0 professionnel**.
N°9151, 2000, 1 150 pages.

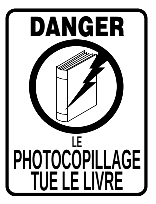
L. LACROIX, N. LEPRINCE, C. BOGGERO, C. LAUER. – **Programmation Web avec PHP**.
N°9113, 2000, 382 pages.

Le livre de Java premier langage

Anne Tasso



ÉDITIONS EYROLLES
61, Bld Saint-Germain
75240 Paris Cedex 05
www.editions-eyrolles.com



Avant-propos

Organisation de l'ouvrage

Ce livre est tout particulièrement destiné aux débutants qui souhaitent aborder l'apprentissage de la programmation en utilisant le langage Java comme premier langage.

Les concepts fondamentaux de la programmation y sont présentés de façon évolutive, grâce à un découpage de l'ouvrage en trois parties, chacune couvrant un aspect différent des outils et techniques de programmation.

Le chapitre introductif, « Naissance d'un programme », constitue le préalable nécessaire à la bonne compréhension des parties suivantes. Il introduit aux mécanismes de construction d'un algorithme, compte tenu du fonctionnement interne de l'ordinateur, et explique les notions de langage informatique, de compilation et d'exécution à travers un exemple de programme écrit en Java.

La première partie concerne l'étude des « Outils et techniques de base » :

- Le chapitre 1, « Stocker une information », aborde la notion de variables et de types. Il présente comment stocker une donnée en mémoire, calculer des expressions mathématiques ou échanger deux valeurs et montre comment le type d'une variable peut influencer sur le résultat d'un calcul.
- Le chapitre 2, « Communiquer une information », explique comment transmettre des valeurs à l'ordinateur par l'intermédiaire du clavier et montre comment l'ordinateur fournit des résultats en affichant des messages à l'écran.
- Le chapitre 3, « Faire des choix », examine comment tester des valeurs et prendre des décisions en fonction du résultat. Il traite de la comparaison de valeurs ainsi que de l'arborescence de choix.
- Le chapitre 4, « Faire des répétitions », est consacré à l'étude des outils de répétition et d'itération. Il aborde les notions d'incrémentation et d'accumulation de valeurs (compter et faire la somme d'une collection de valeurs).

La deuxième partie, « Initiation à la programmation orientée objet », introduit les concepts fondamentaux indispensables à la programmation objet.

- Le chapitre 5, « De l'algorithme paramétré à l'écriture de fonctions », montre l'intérêt de l'emploi de fonctions dans la programmation. Il examine les différentes étapes de leur création.
- Le chapitre 6, « Fonctions, notions avancées », décrit très précisément comment manipuler les fonctions et leurs paramètres. Il définit les termes de variable locale et de classe et explique le passage de paramètres par valeur.
- Le chapitre 7, « Les classes et les objets », explique, à partir de l'étude de la classe `String`, ce que sont les classes et les objets dans le langage Java. Il montre ensuite comment définir de nouvelles classes et construire des objets propres à l'application développée.
- Le chapitre 8, « Les principes du concept d'objet », développe plus particulièrement comment les objets se communiquent l'information, en expliquant notamment le principe du passage de paramètres par référence. Il décrit ensuite les principes fondateurs de la notion d'objet, c'est-à-dire l'encapsulation des données (protection et contrôle des données, constructeur de classe) ainsi que l'héritage entre classes.

La troisième partie, « Outils et techniques orientés objet », donne tous les détails sur l'organisation, le traitement et l'exploitation intelligente des objets.

- Le chapitre 9, « Collectionner un nombre fixe d'objets », concerne l'organisation des données sous la forme d'un tableau de taille fixe.
- Le chapitre 10, « Collectionner un nombre indéterminé d'objets », présente les différents outils qui permettent d'organiser dynamiquement en mémoire les ensembles de données de même nature. Il est également consacré aux différentes techniques d'archivage et à la façon d'accéder aux informations stockées sous forme de fichiers.
- Le chapitre 11, « Dessiner des objets », couvre une grande partie des outils graphiques proposés par le langage Java. Il analyse le concept événement-action et décrit comment réaliser une applet.

Ce livre contient également en annexe :

- Un guide d'utilisation du CD-Rom expliquant comment utiliser les outils proposés dans le CD-Rom.
- Un index, qui vous aidera à retrouver une information sur un thème que vous recherchiez (les mots-clés du langage, les exemples, les principes de fonctionnement, les classes et leurs méthodes, etc.).

Table des matières

INTRODUCTION

Naissance d’un programme

Aujourd’hui, l’informatique en général et l’ordinateur en particulier sont d’un usage courant. Grâce à Internet, l’informatique donne accès à une information mondiale. Elle donne aussi la possibilité de traiter cette information pour analyser, gérer, prévoir ou concevoir des événements dans des domaines aussi divers que la météo, la médecine, l’économie, la bureautique, etc.

Cette communication et ces traitements ne sont possibles qu’au travers de l’outil informatique. Cependant, toutes ces facultés résultent davantage de l’application d’un programme résidant sur l’ordinateur que de l’ordinateur lui-même. En fait, le programme est à l’ordinateur ce que l’esprit est à l’être humain.

Créer une application, c’est apporter de l’esprit à l’ordinateur. Pour que cet esprit donne sa pleine mesure, il est certes nécessaire de bien connaître le langage des ordinateurs, mais, surtout, il est indispensable de savoir programmer. La programmation est l’art d’analyser un problème afin d’en extraire la marche à suivre, l’algorithme susceptible de résoudre ce problème.

C’est pourquoi ce chapitre commence par aborder la notion d’algorithme. À partir d’un exemple tiré de la vie courante, nous déterminons les étapes essentielles à l’élaboration d’un programme (*« Construire un algorithme »*). À la section suivante, « Qu’est-ce qu’un ordinateur ? », nous examinons le rôle et le fonctionnement de l’ordinateur dans le passage de l’algorithme au programme. Nous étudions ensuite, à travers un exemple simple, comment écrire un programme en Java et l’exécuter (*« Un premier programme en Java, ou comment parler à un ordinateur »*). Enfin, nous décrivons, à la section Le projet « Gestion d’un compte bancaire », le cahier des charges de l’application projet que le lecteur assidu peut réaliser en suivant les exercices décrits à la fin de chaque chapitre.

Construire un algorithme

Un ordinateur muni de l’application adéquate traite une information. Il sait calculer, compter, trier ou rechercher l’information, dans la mesure où un programmeur lui a donné les ordres à exécuter et la marche à suivre pour arriver au résultat.

Cette marche à suivre s’appelle un algorithme.

Déterminer l'algorithme, c'est trouver un cheminement de tâches à fournir à l'ordinateur pour qu'il les exécute. Voyons comment s'y prendre pour construire cette marche à suivre.

Ne faire qu'une seule chose à la fois

Avant de réaliser une application concrète, telle que celle proposée en projet dans cet ouvrage, nécessairement complexe par la diversité des tâches qu'elle doit réaliser, simplifions-nous la tâche en ne cherchant à résoudre qu'un problème à la fois.

Considérons que créer une application, c'est décomposer cette dernière en plusieurs sous-applications qui, à leur tour, se décomposent en micro-applications, jusqu'à descendre au niveau le plus élémentaire. Cette démarche est appelée **analyse descendante**. Elle est le principe de base de toute construction algorithmique.

Pour bien comprendre cette démarche, penchons-nous sur un problème réel et simple à résoudre : comment faire un café chaud non sucré ?

Exemple : l'algorithme du café chaud

Construire un algorithme, c'est avant tout analyser l'énoncé du problème afin de définir l'ensemble des objets à manipuler pour obtenir un résultat.

Définition des objets manipulés

Analysons l'énoncé suivant :

```
Comment faire un café chaud non sucré ?
```

Chaque mot a son importance, et « non sucré » est aussi important que « café » ou « chaud ». Le terme « non sucré » implique qu'il ne soit pas nécessaire de prendre du sucre ni une petite cuillère.

Remarquons que tous les ingrédients et ustensiles nécessaires ne sont pas cités dans l'énoncé. En particulier, nous ne savons pas si nous disposons d'une cafetière électrique ou non. Pour résoudre notre problème, nous devons prendre certaines décisions, et ces dernières vont avoir une influence sur l'allure générale de notre algorithme.

Supposons que, pour réaliser notre café, nous soyons en possession des ustensiles et ingrédients suivants :

```
café moulu
filtre
eau
pichet
cafetière électrique
tasse
électricité
table
```

En fixant la liste des ingrédients et des ustensiles, nous définissons un environnement, une base de travail. Nous sommes ainsi en mesure d'établir une liste de toutes les actions à mener pour résoudre le problème et de construire la marche à suivre permettant d'obtenir un café.

Liste des opérations

```
Verser l'eau dans la cafetière, le café dans la tasse, le café dans le filtre.
Remplir le pichet d'eau.
Prendre du café moulu, une tasse, de l'eau, une cafetière électrique, un
➥filtre, le pichet de la cafetière.
Brancher, allumer ou éteindre la cafetière électrique.
Attendre que le café remplisse le pichet.
Poser la tasse, la cafetière sur la table, le filtre dans la cafetière,
➥le pichet dans la cafetière.
```

Cette énumération est une description de toutes les actions nécessaires à la réalisation d'un café chaud.

Chaque action est un fragment du problème donné et ne peut plus être découpée. Chaque action est élémentaire par rapport à l'environnement que nous nous sommes donné.

En définissant l'ensemble des actions possibles, nous créons un langage minimal qui nous permet de réaliser le café. Ce langage est composé de verbes (`Prendre`, `Poser`, `Verser`, `Faire`, `Attendre…`) et d'objets (`Café moulu`, `Eau`, `Filtre`, `Tasse…`).

La taille du langage, c'est-à-dire le nombre de mots qu'il renferme, est déterminée par l'environnement. Pour cet exemple, nous avons, en précisant les hypothèses, volontairement choisi un environnement restreint. Nous aurions pu décrire des tâches comme « prendre un contrat EDF » ou « planter une graine de café », mais elles ne sont pas utiles à notre objectif pédagogique.

Telle que nous l'avons décrite, la liste des opérations ne nous permet pas encore de faire un café chaud. En suivant cette liste, tout y est, mais dans le désordre. Pour réaliser ce fameux café, nous devons ordonner cette liste.

Ordonner la liste des opérations

```
1.   Prendre une cafetière électrique.
2.   Poser la cafetière sur la table.
3.   Prendre un filtre.
4.   Poser le filtre dans la cafetière.
5.   Prendre du café moulu.
6.   Verser le café moulu dans le filtre.
7.   Prendre le pichet de la cafetière.
8.   Remplir le pichet d'eau.
9.   Verser l'eau dans la cafetière.
10.  Poser le pichet dans la cafetière.
11.  Brancher la cafetière.
12.  Allumer la cafetière.
13.  Attendre que le café remplisse le pichet.
14.  Prendre une tasse.
15.  Poser la tasse sur la table.
16.  Eteindre la cafetière.
17.  Prendre le pichet de la cafetière.
18.  Verser le café dans la tasse.
```

L'exécution de l'ensemble ordonné de ces tâches nous permet maintenant d'obtenir du café chaud non sucré.

Remarquons que l'ordre d'exécution de cette marche à suivre est important. En effet, si l'utilisateur réalise l'opération 10 (`Allumer la cafetière`) avant l'opération 8 (`Verser l'eau dans la cafetière`), le résultat est sensiblement différent. La marche à suivre ainsi désordonnée risque de détériorer la cafetière électrique.

Cet exemple tiré de la vie courante montre que, pour résoudre un problème, il est essentiel de définir les objets utilisés puis de trouver la suite logique de tous les ordres nécessaires à la résolution dudit problème.

Vers une méthode

La tâche consistant à décrire comment résoudre un problème n'est pas simple. Elle dépend en partie du niveau de difficulté du problème et réclame un savoir-faire : la façon de procéder pour découper un problème en actions élémentaires.

Pour aborder dans les meilleures conditions possibles la tâche difficile d'élaboration d'un algorithme, nous devons tout d'abord :

- déterminer les objets utiles à la résolution du problème ;
- construire et ordonner la liste de toutes les actions nécessaires à cette résolution.

Pour cela, il est nécessaire :

- d'analyser en détail la tâche à résoudre ;
- de fractionner le problème en actions distinctes et élémentaires.

Ce fractionnement est réalisé en tenant compte du choix des hypothèses de travail. Ces hypothèses imposent un ensemble de contraintes, qui permettent de savoir si l'action décrite est élémentaire et peut ne plus être découpée.

Cela fait, nous avons construit un algorithme.

Passer de l'algorithme au programme

Pour construire un algorithme, nous avons défini des hypothèses de travail, c'est-à-dire supposé une base de connaissances minimales nécessaires à la résolution du problème. Ainsi, le fait de prendre l'hypothèse d'avoir du café moulu nous autorise à ne pas décrire l'ensemble des tâches précédant l'acquisition du café moulu. C'est donc la connaissance de l'environnement de travail qui détermine en grande partie la construction de l'algorithme.

Pour passer de l'algorithme au programme, le choix de l'environnement de travail n'est plus de notre ressort. Jusqu'à présent, nous avons supposé que l'exécutant était humain. Maintenant, notre exécutant est l'ordinateur. Pour écrire un programme, nous devons savoir ce dont est capable un ordinateur et connaître son fonctionnement de façon à établir les connaissances et capacités de cet exécutant.

Qu'est-ce qu'un ordinateur ?

Notre intention n'est pas de décrire en détail le fonctionnement de l'ordinateur et de ces composants mais d'en donner une image simplifiée.

Pour tenter de comprendre comment travaille l’ordinateur et, surtout, comment il se programme, nous allons schématiser à l’extrême ses mécanismes de fonctionnement.

Un ordinateur est composé de deux parties distinctes, la **mémoire centrale** et l’**unit**é **centrale.**

La mémoire centrale permet de mémoriser toutes les informations nécessaires à l’exécution d’un programme. Ces informations correspondent à des **données** ou à des ordres à exécuter (**instructions**). Les ordres placés en mémoire sont effectués par l’unité centrale, la partie active de l’ordinateur.

Lorsqu’un ordinateur exécute un programme, son travail consiste en grande partie à gérer la mémoire, soit pour y lire une instruction, soit pour y stocker une information. En ce sens, nous pouvons voir l’ordinateur comme un robot qui sait agir en fonction des ordres qui lui sont fournis. Ces actions, en nombre limité, sont décrites ci-dessous.

Déposer ou lire une information dans une case mémoire

La mémoire est formée d’éléments, ou cases, qui possèdent chacune un numéro (une adresse). Chaque case mémoire est en quelque sorte une boîte aux lettres pouvant contenir une information (une lettre). Pour y déposer cette information, l’ordinateur (le facteur) doit connaître l’adresse de la boîte. Lorsque le robot place une information dans une case mémoire, il mémorise l’adresse où se situe celle-ci afin de retrouver l’information en temps nécessaire.

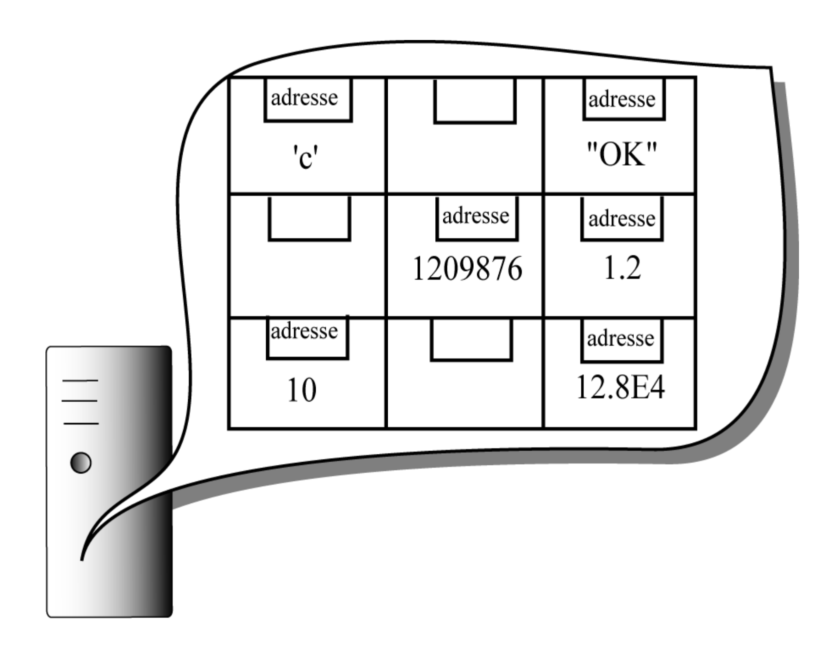


Figure I–1.

La mémoire de l’ordinateur est composée de cases possédant une adresse et pouvant contenir à tout moment une valeur.

Le robot sait déposer une information dans une case, mais il ne sait pas la retirer (au sens de prendre un courrier déposé dans une boîte aux lettres). Lorsque le robot prend l’information déposée dans une case mémoire, il ne fait que la lire. En aucun cas il ne la retire ni ne l’efface. L’information lue reste toujours dans la case mémoire.

Pour effacer une information d’une case mémoire, il est nécessaire de placer une nouvelle information dans cette même case. Ainsi, la nouvelle donnée remplace l’ancienne, et l’information précédente est détruite.

Exécuter des opérations simples telles que l'addition ou la soustraction

Le robot lit et exécute les opérations dans l'ordre où elles lui sont fournies. Pour faire une addition, il va chercher les valeurs à additionner dans les cases mémoire appropriées (stockées, par exemple, aux adresses `a` et `b`) et réalise ensuite l'opération demandée. Il enregistre alors le résultat de cette opération dans une case d'adresse `c`. De telles opérations sont décrites à l'aide d'ordres, appelés aussi **instructions**.

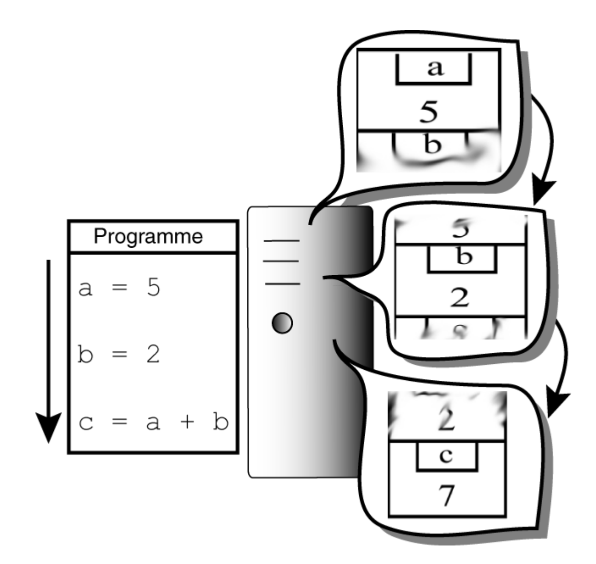


Figure I–2.
Le programme exécute les instructions dans l'ordre de leur apparition.

Comparer des valeurs

Le robot est capable de comparer deux valeurs entre elles pour déterminer si l'une d'entre elle est plus grande, plus petite, égale ou différente de l'autre valeur. Grâce à la comparaison, le robot est capable de tester une condition et d'exécuter un ordre plutôt qu'un autre, en fonction du résultat du test.

La réalisation d'une comparaison ou d'un test fait que le robot ne peut plus exécuter les instructions dans leur ordre d'apparition. En effet, suivant le résultat du test, il doit rompre l'ordre de la marche à suivre, en sautant une ou plusieurs instructions. C'est pourquoi il existe des instructions particulières dites de **branchement**. Grâce à ce type d'instructions, le robot est à même non seulement de sauter des ordres mais aussi de revenir à un ensemble d'opérations afin de les répéter.

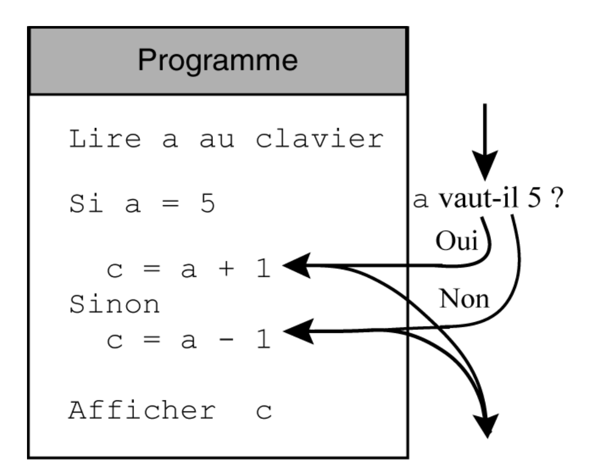


Figure I–3.
Suivant le résultat du test, l'ordinateur exécute l'une ou l'autre instruction en sautant celle qu'il ne doit pas exécuter.

Communiquer une information élémentaire

Un programme est essentiellement un outil qui traite l'information. Cette information est transmise à l'ordinateur par l'utilisateur. L'information est saisie par l'intermédiaire du clavier ou de la souris. Cette transmission de données à l'ordinateur est appelée communication d'entrée (*input* en anglais). On parle aussi de **saisie** ou encore de **lecture** de données.

Après traitement, le programme fournit un résultat à l'utilisateur, soit par l'intermédiaire de l'écran, soit sous forme de fichiers, que l'on peut ensuite imprimer.

Il s'agit alors de communication de sortie (*output*) ou encore d'**affichage** ou d'**écriture** de données.

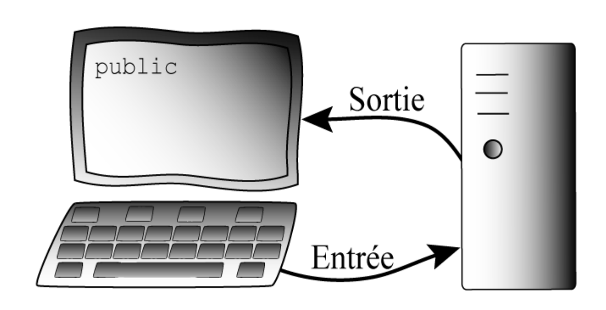


Figure I–4..
La saisie au clavier d'une valeur correspond à une opération d'entrée, et l'affichage d'un résultat à une opération de sortie.

Coder l'information

De par la nature de ses composants électroniques, le robot ne perçoit que deux états : composant allumé et composant éteint. De cette perception découle le langage binaire, qui utilise par convention les deux symboles 0 (éteint) et 1 (allumé).

Ne connaissant que le 0 et le 1, l'ordinateur utilise un code pour représenter une information aussi simple qu'un nombre entier ou un caractère. Ce code est un programme, qui différencie chaque type d'information et transforme une information (donnée numérique ou alphabétique) en valeurs binaires. À l'inverse, ce programme sait aussi transformer un nombre binaire en valeur numérique ou alphabétique. Il existe autant de codes que de types d'informations. Cette différenciation du codage (en fonction de ce qui doit être représenté) introduit le concept de **type** de données.

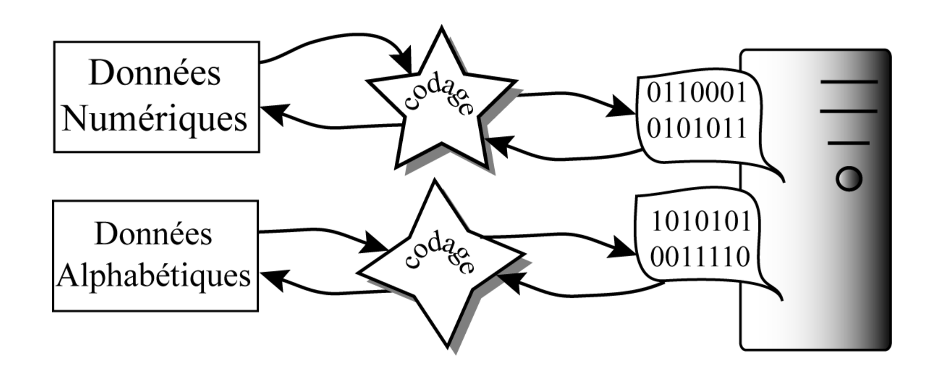


Figure I–5.
Toute information est codée en binaire. Il existe autant de codes que de types d'informations .

Signalons en outre que toute information fournie à l'ordinateur est, au bout du compte, codée en binaire. L'information peut être un simple nombre ou une instruction de programme.

Exemple

Pour additionner deux nombres, l'ordinateur fait appel aux trois éléments qui lui sont nécessaires pour réaliser cette opération. Ces éléments sont les suivants :

- Le code binaire représentant l'opération d'addition (par exemple 0101).
- L'adresse de la case mémoire où est stocké le premier nombre (par exemple 011101).
- L'adresse de la case mémoire où se trouve la deuxième valeur (par exemple 010101).

Pour finir, l'instruction d'addition de ces deux nombres s'écrit en assemblant les trois codes binaires (soit, dans notre exemple, 0101011101010101).

Remarquons que le code binaire associé à chaque code d'opération (addition, test, etc.) n'est pas nécessairement identique d'un ordinateur à un autre. Ce code binaire est déterminé par le constructeur de l'ordinateur. De ce fait, une instruction telle que l'addition de deux nombres n'a pas le même code binaire d'une machine à une autre. Il existe donc, pour un même programme, un code binaire qui diffère suivant le type d'ordinateur utilisé.

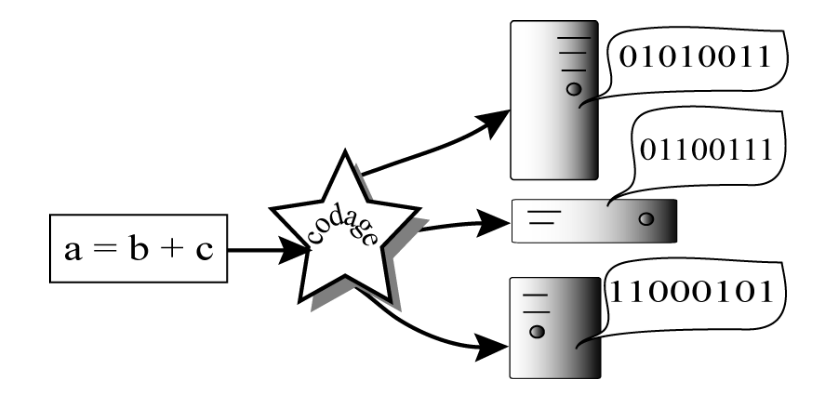


Figure I–6.
Pour un même programme, le code binaire diffère en fonction de l'ordinateur utilisé.

L'ordinateur n'est qu'un exécutant

En pratique, le robot est très habile à réaliser l'ensemble des tâches énoncées ci-dessus. Il les exécute beaucoup plus rapidement qu'un être humain.

En revanche, le robot n'est pas doué d'intelligence. Il n'est ni capable de choisir une action plutôt qu'une autre, ni apte à exécuter de lui-même l'ensemble de ces actions. Pour qu'il puisse exécuter une instruction, il faut qu'un être humain détermine l'instruction la plus appropriée et lui donne l'ordre de l'exécuter.

Le robot est un exécutant capable de comprendre des ordres. Compte tenu de ses capacités limitées, les ordres ne peuvent pas lui être donnés dans le langage naturel propre à l'être humain. En effet, le robot ne comprend pas le sens des ordres qu'il exécute mais seulement leur forme. Chaque ordre doit être écrit avec des mots particuliers et une forme, ou syntaxe, préétablie. L'ensemble de ces mots constitue un langage informatique. Les langages C, C++, Pascal, Basic, Fortran, Cobol et Java sont des langages de programmation, constitués de mots et d'ordres dont la syntaxe diffère selon le langage.

Pour écrire un programme, il est nécessaire de connaître un de ces langages, de façon à traduire un algorithme en un programme composé d'ordres.

Un premier programme en Java, ou comment parler à un ordinateur

Pour créer une application, nous allons avoir à décrire une liste ordonnée d'opérations dans un langage compréhensible par l'ordinateur. La contrainte est de taille et se porte essentiellement sur la façon de définir et de représenter les objets nécessaires à la résolution du problème en fonction du langage de l'ordinateur.

Pour bien comprendre la difficulté du travail à accomplir, regardons comment faire calculer à un ordinateur la circonférence d'un cercle de rayon quelconque.

Calcul de la circonférence d'un cercle

L'exercice consiste à calculer le périmètre d'un cercle de rayon quelconque. Nous supposons que l'utilisateur emploie le clavier pour transmettre au programme la valeur du rayon.

Définition des objets manipulés

Pour calculer la circonférence du cercle, l'ordinateur a besoin de stocker dans ses cases mémoire la valeur du rayon ainsi que celle du périmètre. Les objets à manipuler sont deux valeurs numériques appartenant à l'ensemble des réels $\mathbb{R}$. Nous appelons P la valeur correspondant au périmètre et R la valeur du rayon.

La liste des opérations

La circonférence d'un cercle est calculée à partir de la formule : $P = 2 \times \pi \times R$.

La valeur du rayon est fournie par l'utilisateur à l'aide du clavier. Elle n'est donc pas connue au moment de l'écriture du programme. En conséquence, il est nécessaire d'écrire l'ordre (instruction) de saisie au clavier avant de calculer la circonférence.

La liste des opérations est la suivante :

```
1. Réserver deux cases mémoire pour y stocker les valeurs correspondant au rayon
   (R) et au périmètre (P).
2. Demander à l'utilisateur de saisir la valeur du rayon au clavier et la
   placer dans la case mémoire associée.
3. Connaissant la valeur du rayon, calculer la circonférence.
4. Afficher le résultat.
```

La valeur du rayon puis, après calcul, celle de la circonférence sont les données principales de ce programme. L'ordinateur doit les stocker en mémoire pour les utiliser.

L'opération `1` consiste à donner un nom aux cases mémoire qui vont servir à stocker ces données. Lors de cette opération, appelée **déclaration de variables,** l'ordinateur réserve une case mémoire pour chaque nom de variable défini. Ici, ces variables ont pour nom `P` et `R`. Au cours de cette réservation d'emplacements mémoire, l'ordinateur associe le nom de la variable et l'adresse réelle de la case mémoire.

Pour le programmeur, le nom et l'adresse d'une case ne font qu'un, car il ne manipule les variables que par leur nom, alors que l'ordinateur travaille avec leur adresse. En donnant un nom à une case, l'être humain sait facilement identifier les objets qu'il manipule, alors qu'il lui serait pénible de manipuler les adresses binaires correspondantes.

Inversement, en associant un nom à une adresse codée en binaire, l'ordinateur peut véritablement manipuler ces objets.

L'opération 2 permet de saisir au clavier la valeur du rayon. Pour que l'utilisateur non initié sache à quoi correspond la valeur saisie, il est nécessaire, avant de procéder à cette saisie, d'afficher un message explicatif à l'écran. L'opération 2 se décompose en deux instructions élémentaires, à savoir :

```
Afficher un message demandant à l'utilisateur du programme de saisir une valeur
➥pour le rayon
Une fois la valeur saisie par l'utilisateur, la placer dans sa case mémoire
```

Les opérations 3 et 4 sont des actions élémentaires directement traduisibles en langage informatique.

La traduction en Java

Une application, ou programme, ne s'écrit pas en une seule fois. Nous verrons à la lecture de cet ouvrage que programmer c'est toujours décomposer une difficulté en différentes tâches plus aisées à réaliser. Cette décomposition s'applique aussi bien pour construire un algorithme que pour l'écriture du programme lui-même.

D'une manière générale, la meilleure façon de procéder pour fabriquer un programme revient à écrire une première ébauche et à la tester. De ces tests, il ressort des fautes à corriger et, surtout, de nouvelles idées. Le programme final consiste en l'assemblage de toutes ces corrections et de ces améliorations.

Pour traduire la marche à suivre définie précédemment selon les règles de syntaxe du langage Java, nous allons utiliser cette même démarche. Nous nous intéresserons, dans un premier temps, à la traduction du cœur du programme (opérations 1 à 4 décrites à la section précédente). Nous verrons pour finir comment insérer l'ensemble de ces instructions dans une structure de programme Java.

- L'opération 1 consiste à déclarer les variables utilisées pour le calcul de la circonférence. Cette opération se traduit par l'instruction :

```
double R, P ;
```

  Par cette instruction, le programme demande à l'ordinateur de réserver deux cases mémoire, nommées R et P, pour y stocker les valeurs du rayon et de la circonférence. Le mot réservé double permet de préciser que les valeurs numériques sont réelles « avec une double précision », c'est-à-dire avec une précision pouvant aller jusqu'à 17 chiffres après la virgule.

  ✔ Pour plus d'informations sur la définition des types de variables, reportez-vous au chapitre 1, « Stocker une information ».

- Pour réaliser l'opération 2, nous devons faire afficher un message demandant à l'utilisateur de saisir une valeur. Cette opération se traduit par l'instruction :

```
System.out.print("Valeur du rayon : ") ;
```

  System.out.print() est ce que l'on appelle un programme, ou une fonction, prédéfini

par le langage Java. Ce programme permet d'écrire à l'écran le message spécifié à l'intérieur des parenthèses. Le message affiché est ici un fragment de texte, appelé, dans le jargon informatique, une chaîne de caractères. Pour que l'ordinateur comprenne que la chaîne de caractères n'est pas un nom de variable mais un texte à afficher, il faut placer entre guillemets (`" "`) tous les caractères composant la chaîne.

- L'opération `2` est terminée lorsque la valeur demandée est effectivement saisie et stockée en mémoire. Pour ce faire, nous devons écrire l'instruction suivante :

```
R = Lire.d() ;
```

`Lire.d()`est un programme proposé par l'auteur, qui permet de communiquer une valeur numérique au programme par l'intermédiaire du clavier. Ce programme est une fonction qui donne l'ordre à l'ordinateur d'attendre la saisie d'une valeur de double précision. La saisie est effective lorsque l'utilisateur valide sa réponse en appuyant sur la touche `entrée` du clavier. Cette fonction n'étant pas prédéfinie par le langage Java, elle figure dans le CD-Rom livré avec le manuel.

Une fois la valeur saisie, elle est placée dans la variable `R` grâce au signe `=`.

> ✔ Pour plus de précisions sur les deux méthodes `System.out.print()` et `Lire.d()`, reportez-vous au chapitre 2, « Communiquer une information ». Pour le signe =, voir le chapitre 1, « Stocker une information ».

- L'opération `3` permet de calculer la valeur de la circonférence. Elle se traduit de la façon suivante :

```
P = 2 * Math.PI * R ;
```

Le signe `*` est le symbole qui caractérise l'opération de multiplication. `Math.PI` est le terme qui représente la valeur numérique du nombre π avec une précision de 17 chiffres après la virgule. Le mot-clé `Math` désigne la bibliothèque de mathématiques accompagnant le langage Java. Cette bibliothèque contient, outre des constantes telles que π, des fonctions standards, comme `sqrt()` (racine carrée) ou `sin()` (sinus). Une fois les opérations de multiplication effectuées, la valeur calculée est placée dans la variable `P` grâce au signe `=`.

- La dernière opération (`4`) de notre programme a pour rôle d'afficher le résultat du calcul précédent. Cet affichage est réalisé grâce à l'instruction :

```
System.out.print("Le cercle de rayon " + R + " a pour perimetre : " + P);
```

Ce deuxième appel à la fonction `System.out.print()` est plus complexe que le premier. Il mélange l'affichage de chaînes de caractères (texte entre guillemets) et de contenu de variables.

Si les caractères `R` et `P` ne sont pas placés entre guillemets, c'est pour que l'ordinateur les interprète non pas comme des caractères à afficher mais comme les variables qui ont été déclarées en début de programme. De ce fait, il affiche le contenu des variables et non les lettres R et P.

Les signes +, qui apparaissent dans l'expression `"Le cercle de rayon " + R + " a pour perimetre : " + P"`, indiquent que chaque élément du message doit être affiché en le collant aux autres : d'abord la chaîne de caractères `"Le cercle de rayon "`, puis la valeur de `R`, puis la chaîne `"a pour périmètre :"` et, pour finir, la valeur de `P`.

En résumé, la partie centrale du programme contient les cinq instructions suivantes :

```
double R, P ;
System.out.print("Valeur du rayon : ") ;
R = Lire.d() ;
P = 2 * Math.PI * R ;
System.out.print("Le cercle de rayon " + R + " a pour perimetre : " + P);
```

Pour améliorer la lisibilité du programme, il est possible d'insérer dans le programme, des commentaires, comme suit :

```
// Déclaration des variables
double R, P ;
// Afficher le message "Valeur du rayon : " à l'écran
System.out.print("Valeur du rayon : ") ;
// Lire au clavier une valeur, placer cette valeur dans la variable R
R = Lire.d() ;
// Calculer la circonférence en utilisant la formule consacrée
P = 2 * Math.PI * R ;
// Afficher le résultat
System.out.print("Le cercle de rayon " + R + " a pour perimetre : " + P);
```

Les lignes du programme qui débutent par les signes `//` sont considérées par l'ordinateur non pas comme des ordres à exécuter mais comme des lignes de commentaire. Elles permettent d'expliquer en langage naturel ce que réalise l'instruction associée.

Écrites de la sorte, ces instructions constituent le cœur de notre programme. Elles ne peuvent cependant pas encore être interprétées correctement par l'ordinateur. En effet, celui-ci exécute les instructions d'un programme dans l'ordre de leur arrivée. Une application doit donc être constituée d'une instruction qui caractérise le début du programme. Pour ce faire, nous devons écrire notre programme ainsi :

```
public static void main(String [] argument)
{
 // Déclaration des variables
 double R, P ;
 // Afficher le message "Valeur du rayon : " à l'écran
 System.out.print("Valeur du rayon : ") ;
 // Lire au clavier une valeur, placer cette valeur dans la variable R
 R = Lire.d() ;
 // Calculer la circonférence en utilisant la formule consacrée
 P = 2 * Math.PI * R ;
 // Afficher le résultat
 System.out.print("Le cercle de rayon " + R + " a pour perimetre : "+ P) ;
} // Fin de la fonction main()
```

La ligne `public static void main(String [] argument)` est l'instruction qui permet d'indiquer à l'ordinateur le début du programme. Ce début est identifié par ce que l'on appelle la fonction `main()`, c'est-à-dire la fonction principale du programme. De cette façon, lorsque le programme est exécuté, l'ordinateur recherche le mot-clé `main`. Une fois ce mot-clé trouvé, l'ordinateur exécute une à une chaque instruction constituant la fonction.

Les autres mots-clés, tels que `public`, `static` ou `void`, déterminent certaines caractéristiques de la fonction `main()`. Ces mots-clés, obligatoirement placés et écrits dans cet ordre, sont expliqués au fur et à mesure de leur apparition dans le livre et plus particulièrement à la section « Quelques techniques utiles » du chapitre « Collectionner un nombre fixe d'objets ».

Pour finir, nous devons insérer la fonction `main()` dans ce qui est appelé une classe Java. En programmation objet, un programme n'est exécutable que s'il est défini à l'intérieur d'une classe. Une classe est une entité interprétée par l'ordinateur comme étant une unité de programme, qu'il peut exécuter dès qu'un utilisateur le souhaite.

Aucun programme ne peut être écrit en dehors d'une classe. Nous devons donc placer la fonction `main()` à l'intérieur d'une classe définie par l'instruction `public class Cercle {}`, comme suit :

```
public class Cercle
{
 public static void main(String [] argument)
 {
  // Déclaration des variables
  double R, P ;
  // Afficher le message "Valeur du rayon : " à l'écran
  System.out.print("Valeur du rayon : ") ;
  // Lire au clavier une valeur, placer cette valeur dans la variable R
  R = Lire.d() ;
  // Calculer la circonférence en utilisant la formule consacrée
  P = 2*Math.PI*R ;
  // Afficher le résultat
  System.out.print("Le cercle de rayon "+ R +" a pour perimetre : "+ P);
 }
} // Fin de la classe Cercle
```

Nous obtenons ainsi le programme dans son intégralité. La ligne `public class Cercle` permet de définir une classe. Puisque notre programme effectue des opérations sur un cercle, nous avons choisi d'appeler cette classe `Cercle`. Nous aurions pu lui donner un tout autre nom, comme `Rond` ou `Exemple`. Ainsi définie, la classe `Cercle` devient un programme à part entière


Pour voir le résultat de l'exécution de ce programme, reportez-vous à la section « Exemple sur plate-forme Unix », ci-après.

Figure I–7 .

Un programme Java est constitué de deux blocs encastrés. Le premier bloc représente la classe associée au programme, tandis que le second détermine la fonction principale.

```
public class Cercle
{
    public static void main( String [] arg)
    {

    }
}
```

En observant la Figure 7, nous remarquons que ce programme, de même que tous ceux à venir, est constitué de deux blocs encastrés définis par les deux lignes `public class Cercle{}` et `public static void main(String [] argument){}`.

Ces deux blocs constituent la charpente principale et nécessaire à tout programme écrit avec le langage Java. Cet exemple montre en outre que les mots réservés par le langage Java sont nombreux et variés et qu'ils constituent une partie du langage Java.

Si la syntaxe, c'est-à-dire la forme, de ces instructions peut paraître étrange de prime abord, nous verrons à la lecture de cet ouvrage que leur emploi obéit à des règles strictes. En apprenant ces règles et en les appliquant, vous pourrez vous initier aux techniques de construction d'un programme, qui reviennent à décomposer un problème en actions élémentaires puis à traduire celles-ci à l'aide du langage Java.

Exécuter un programme

Nous avons écrit un programme constitué d'ordres, dont la syntaxe obéit à des règles strictes. Pour obtenir le résultat des calculs décrits dans le programme, nous devons le faire lire par l'ordinateur, c'est-à-dire l'exécuter.

Pour cela, nous devons traduire le programme en langage machine. En effet, nous l'avons vu, l'ordinateur ne comprend qu'un seul langage, le langage binaire.

Compiler, ou traduire en langage machine

Cette traduction du code source (le programme écrit en langage informatique) en code machine exécutable (le code binaire) est réalisée par un programme appelé **compilateur**. L'opération de compilation consiste à lancer un programme qui lit chaque instruction du code source et vérifie si celles-ci ont une syntaxe correcte. S'il n'y a pas d'erreur, le compilateur crée un code binaire directement exécutable par l'ordinateur.

Il existe autant de compilateurs que de langages. Un programme écrit en langage Pascal est traduit en binaire à l'aide d'un compilateur Pascal, et un programme écrit en Java est compilé par un compilateur Java. Le compilateur Java ne travaille pas tout à fait comme un compilateur classique, traduisant un code source en code exécutable. Pour mieux comprendre cette différence, voyons son fonctionnement et comment l'utiliser.

Compiler un programme écrit en Java

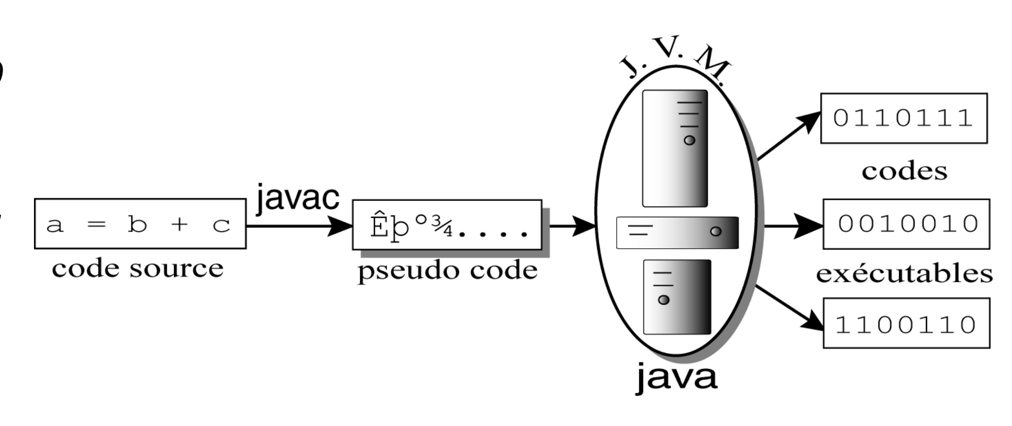
L'objectif premier de J. Gosling, le créateur du langage Java, a été de réaliser un langage indépendant de l'ordinateur. Dans cette optique, un programme écrit sur PC, par exemple, doit pouvoir s'exécuter sur un PC (de type IBM), un Macintosh (Apple) ou une station Unix (de type Sun), et ce sans réécriture ni compilation du code source.

Or, le code binaire est spécifique de chaque machine, comme nous l'avons vu à la section « Coder l'information ». Il est impossible de faire tourner un même programme source d'une machine à une autre sans le compiler à nouveau. En effet, lors de la nouvelle compilation, des erreurs apparaissent, dues aux différences de matériel informatique. Pour corriger cet inconvénient majeur, l'idée de J.Gosling a été de créer un code intermédiaire entre le code source et le code binaire. Ce code intermédiaire est appelé pseudo-code, ou encore byte code.

En effet, en créant un pseudo-code, identique pour tous les ordinateurs, il est possible d'exécuter ce code sur différentes machines, sans avoir à le recompiler. Cette exécution est réalisée par un programme spécifique de la machine utilisée, qui interprète et exécute le pseudo-code, compte tenu des ressources propres de l'ordinateur.

Figure I–8 .

Le compilateur (`javac`) transforme le code source en pseudo-code. Ce dernier est exécuté grâce à un interpréteur (`java`) spécifique de chaque type de machine. L'ensemble des interpréteurs constitue la JVM.



Ce programme s'appelle un interpréteur Java. Il en existe autant que de types d'ordinateurs (plates-formes). L'ensemble des ces interpréteurs constitue ce que l'on appelle la machine virtuelle Java, ou JVM (*Java Virtual Machine*).

Le compilateur Java ne crée pas de code binaire, à la différencedes autres compilateurs, tels que les compilateurs C ou C++. Il fabrique un pseudo-code, qui est ensuite interprété par un programme spécifique de l'ordinateur. Ce dernier programme transforme le pseudo-code en code directement exécutable par l'ordinateur choisi. L'avantage d'un tel système est que le développeur d'applications est certain de créer des programmes totalement compatibles avec les différents ordinateurs du marché sans avoir à réécrire une partie du code.

Le kit de développement Java (JDK)

Le tout premier compilateur Java a été écrit par J. Gosling à l'initiative de Sun, le constructeur de stations de travail sous Unix, au début des années 90.

Aujourd'hui, le compilateur Java est téléchargeable depuis le site Internet de Sun. Il est fourni avec le kit de développement Java (JDK *Java Development Kit* ou encore SDK *Sandard Development Kit*). Cet environnement est disponible sur les ordinateurs de type Solaris, PC sous Windows 95/98 ou NT et Mac OS. Si vous souhaitez installer le JDK sur votre machine, consultez sur le CD-Rom fourni avec l'ouvrage le fichier « outils ».

Le JDK est ce que l'on appelle une boîte à outils de développement d'applications. Cela revient à dire qu'il est constitué d'outils, ou programmes, que l'on utilise sous forme de commande, ou ordre. Pour transmettre une commande à un ordinateur, le programmeur doit saisir le nom de cette commande au clavier, dans une fenêtre spécifique du type d'ordinateur utilisé. Les deux principales commandes à connaître pour cet ouvrage sont les commandes de compilation (`javac`) et d'exécution (`java`).

Exemple sur plate-forme Unix

La marche à suivre est la suivante :

1. Entrez le programme qui calcule la circonférence d'un cercle (*exemple donné à la section « Écrire un programme »*) à l'aide d'un éditeur de texte, c'est-à-dire un logiciel permettant de saisir du texte au clavier. Les éditeurs de texte les plus couramment utilisés sous Unix, sont `vi` et `emacs`.

2. Sauvegardez votre programme en choisissant comme nom de fichier celui qui suit les termes `public class`. Pour notre exemple, nous avons écrit `public class Cercle`. Le fichier est donc à sauvegarder sous le nom `Cercle.java`

3. À partir du CD-Rom fourni avec l'ouvrage, copiez le fichier `Lire.java` dans votre répertoire de travail. La présence de ce fichier est nécessaire pour que l'ordinateur demande la saisie d'une valeur au clavier. Pour plus d'informations, reportez-vous au chapitre 2, « Communiquer une information ».

4. Lancez l'ordre de compilation en saisissant sous Unix la commande :

```
javac Cercle.java
```

La compilation est lancée. Le compilateur exécute sa tâche et compile les fichiers `Cercle.java` et `Lire.java`. Au final, si aucune erreur n'est détectée, le compilateur crée un nouveau fichier, appelé `Cercle.class`, ainsi qu'un fichier `Lire.class`. Ces deux fichiers correspondent au pseudo-code relatif à chacun des programmes compilés.

5. Exécutez le programme en lançant la commande :

```
java Cercle
```

La commande `java` lance le programme, qui interprète le pseudo-code créé à l'étape précédente. Ce programme traduit le pseudo-code dans le code binaire conforme à la machine sur laquelle il est lancé. Après exécution, le résultat obtenu à l'écran est :

```
Valeur du rayon : 5
Le cercle de rayon 5 a pour perimetre : 31.41592653589793
```

où `5` est une valeur entrée au clavier par l'utilisateur.

Pour exécuter un programme, les deux étapes suivantes sont nécessaires :

- La compilation du programme à l'aide de la commande `javac` suivie du nom du programme. Une fois la commande réalisée, le pseudo-code est créé et enregistré dans un fichier, dont le nom correspond au nom du programme suivi de l'extension « `.class` ».
- L'exécution du programme en appelant l'interpréteur au moyen de la commande `java`, suivie du nom du programme (sans extension). Cette commande interprète le fichier « `.class` » créé à l'étape précédente et exécute le programme.

Les environnements de développement

Le JDK fournit un ensemble de commandes pour compiler et interpréter. C'est un environnement courant et facile d'emploi dans le monde Unix. Il l'est beaucoup moins, en revanche, sous Windows 95/98/NT. En effet, l'écriture d'une commande telle que donner l'ordre de compiler un programme ne peut se réaliser qu'en ouvrant une fenêtre « Commandes MS-DOS ».

Un certain nombre d'environnements de programmation permettent cependant d'écrire, de compiler puis d'exécuter de façon conviviale un programme Java. Citons, à titre d'exemples les environnements de travail, tels que le logiciel Kawa (Tek-Tools), sur PC, ou Visual Café (Symantec), sur Macintosh.

Ces logiciels offrent, sous forme d'interface graphique conviviale, un ensemble d'outils de développement d'applications. Les outils les plus utilisés sont, en général, les suivants :

- L'éditeur de texte pour écrire le programme.
- Les menus et boîtes à outils, pour lancer la compilation et l'exécution.
- La fenêtre de compilation, qui affiche les éventuelles erreurs de syntaxe.
- La fenêtre d'exécution, qui affiche les messages et résultats du programme en cours.
- La fenêtre qui visualise les projets en cours, dans le cas d'un programme défini à partir de plusieurs fichiers différents.

✔ Vous trouverez toutes les informations nécessaires au téléchargement de ces interfaces dans le fichier « Outils » présent sur le CD-Rom fourni avec l'ouvrage.

Le projet « Gestion d'un compte bancaire »

Pour vous permettre de mieux maîtriser les différentes notions abordées dans cet ouvrage, nous vous proposons de construire une application plus élaborée que les simples exercices appliqués donnés en fin de chapitre.

Dans ce projet, nous avons volontairement évité l'emploi d'interfaces graphiques. Bien qu'attrayantes, ces dernières sont difficiles à maîtriser pour des débutants en programmation. Le projet consiste à bâtir une application autour du concept de menu interactif. À l'heure du tout-graphique, il n'est pas vain d'apprendre à écrire des menus « texte ». Le fait de passer par cet apprentissage permet d'appréhender toutes les notions fonda-

mentales de la programmation, sans avoir à s'évertuer à étudier la syntaxe de toutes les méthodes de la librairie graphique Java.

Cahier des charges

Il s'agit d'écrire une application interactive qui permet de gérer l'ensemble des comptes bancaires d'une personne. Les fonctionnalités fournies par le programme de gestion de comptes bancaires sont les suivantes :

```
Création, Suppression d'un compte
Affichage d'un compte donné
Saisie d'une ligne comptable pour un compte donné
Calcul de statistiques
Sauvegarde des données (n° de compte, lignes comptables)
```

Niveau 1 : programme interactif sous forme de choix dans un menu

L'exécution du programme affiche le menu suivant :

```
1. Créer un compte
2. Afficher un compte
3. Créer une ligne comptable
4. Sortir
5. De l'aide

Votre choix :
```

L'utilisateur choisit une valeur pour exécuter l'opération souhaitée.

- Si l'utilisateur choisit l'option `1`, les informations à fournir concernent :

```
Le type du compte [Types possibles : Compte courant, joint, épargne]
Le numéro du compte
La première valeur créditée
Le taux de placement dans le cas d'un compte épargne
```

- Si l'utilisateur choisit l'option `2`, le programme affiche les caractéristiques d'un compte (type, valeur courante, taux), ainsi que les dix dernières opérations comptables dans l'ordre des dates où ont été effectuées les opérations.
- Pour l'option `3`, il s'agit de fournir des informations pour créer une ligne comptable. Ces informations sont les suivantes :

```
Le numéro du compte concerné (avec vérification de son existence)
La somme à créditer (valeur positive) ou à débiter (valeur négative)
La date de l'opération
Le motif de l'achat ou de la vente [thèmes possibles : Salaire, Loyer,
➥Alimentation, Divers]
Le mode de paiement[Types possibles : CB, n° du Chèque, Virement]
```

- L'option `4. Sortir` du menu général permet de sortir du programme.
- L'option `5. Aide` du menu général affiche une information relative à chaque option du menu.

Niveau 2 : structure de données optimisée en termes d'utilisation de la mémoire

- Le programme doit pouvoir gérer autant de comptes que nécessaire. Pour chaque compte, le nombre d'opérations comptables doit être infini et est donc indéterminable au moment de l'écriture du programme.
- En conséquence, la réservation des cases mémoire ne peut pas être réalisée de façon définitive en tout début de programme. À chaque ligne comptable et à chaque nouveau compte créés, le programme doit être capable de réserver lui-même le nombre suffisant d'emplacements mémoire pour la bonne marche du programme. Lorsque le programme gère lui-même la réservation des emplacements mémoire, on dit qu'il gère sa mémoire de manière dynamique.
- L'option permettant la suppression d'un compte est dépendante de la façon dont est stockée l'information. Cette option ne peut être abordée avant d'avoir choisi le mode de gestion des emplacements mémoire.
- L'option `5. Sortir` du menu général doit contrôler la sauvegarde de l'information. Les données sont sauvegardées sur disque sous forme d'un fichier portant le nom `compte.dat`.

Niveau 3 : s'initier aux graphiques

Un nouveau choix est ajouté à l'option `2. Afficher un compte` du menu général : il s'agit d'afficher les statistiques pour un compte donné sous différentes formes graphiques (histogramme, camembert, etc.).

Les objets manipulés

Un compte bancaire est défini par un ensemble de données :

- un numéro du compte ;
- un type de compte (courant, épargne, joint, etc.) ;
- des lignes comptables possédant chacune une valeur, une date, un thème et un moyen de paiement.

Ces données peuvent être représentées de la façon suivante :

Données	Exemple	Type de l'objet
Numéro du compte	4010.205.530	Caractère
Type du compte	Courant	Caractère
Valeur	– 1 520,30	Numérique
Date	04 03 1978	Date
Thème	Loyer	Caractère
Moyen de paiement	CB	Caractère

Nous verrons, au chapitre 1, « Stocker une information », puis tout au long du chapitre 7, « Les classes et les objets », comment définir et représenter les objets utiles et nécessaires à la réalisation de cette application.

La liste des ordres

Pour créer une application de gestion de comptes bancaires, nous devons décomposer l'ensemble de ses fonctionnalités en tâches élémentaires. Pour ce faire, nous partageons l'application en trois niveaux, de difficulté croissante. Les niveaux 1 et 2 doivent être abordés dans cet ordre et sont nécessaires à la réalisation du niveau 3. La mise en œuvre du niveau 1 permet de réaliser les actions suivantes :

- construire un menu (*voir les chapitres 2, « Communiquer une information », et 3, « Faire des choix »*) ;
- créer des comptes différents ou saisir plusieurs lignes comptables (*voir les chapitres 4, « Faire des répétitions », et 9, « Collectionner un nombre fixe d'objets »*) ;
- définir les comptes et les lignes comptables comme des objets informatiques, au sens de la programmation objet *(voir les chapitre 5, « De l'algorithme paramétré à l'écriture d'une fonction », et 7, « Les classes et les objets »).*

Pour résoudre le niveau 2, nous allons apprendre les tâches suivantes :

- gérer la mémoire de l'ordinateur (*voir les chapitres 9, « Collectionner un nombre fixe d'objets », et 10, « Collectionner un nombre indéterminé d'objets »*) ;
- sauvegarder des informations pour que celles-ci ne disparaissent pas une fois l'ordinateur éteint (*voir le chapitre 10, « Collectionner un nombre indéterminé d'objets »*).

Le niveau 3 va nous initier aux opérations suivantes :

- calculer des statistiques (*voir les chapitres 1, « Stocker une information » et 9, « Collectionner un nombre fixe d'objets »*) ;
- dessiner, en particulier des histogrammes (*voir le chapitre 11, « Dessiner des objets »*).

L'étude étape par étape de l'ensemble de cet ouvrage va nous permettre de réaliser cette application.

Résumé

En informatique, résoudre un problème c'est trouver la suite logique de tous les ordres nécessaires à la solution dudit problème. Cette suite logique est appelée **algorithme**.

La construction d'un algorithme passe par l'analyse du problème, avec pour objectif de le découper en une succession de tâches simplifiées et distinctes. Ainsi, à partir de l'énoncé clair, précis et écrit en français d'un problème, nous devons accomplir les deux opérations suivantes :

- Décomposer l'énoncé en étapes distinctes qui conduisent à l'algorithme.
- Définir les objets manipulés par l'algorithme.

Une fois l'algorithme construit, il faut « écrire le programme », c'est-à-dire **traduire** l'algorithme de façon qu'il soit compris par l'ordinateur. En effet, un programme, c'est un algorithme traduit dans un langage compréhensible par les ordinateurs.

Un ordinateur est composé des deux éléments principaux suivants :

- La **mémoire centrale,** qui sert à mémoriser des ordres ainsi que des informations manipulées par le programme. Schématiquement, on peut dire qu'elle est composée d'emplacements repérés chacun par un nom (côté programmeur) et par une adresse (côté ordinateur).
- L'**unité centrale**, qui exécute une à une les instructions du programme dans leur ordre de lecture. Elle constitue la partie active de l'ordinateur. Ces actions, en nombre limité, sont les suivantes :
  – déposer une information dans une case mémoire ;
  – exécuter des opérations simples, telles que l'addition, la soustraction, etc. ;
  – comparer des valeurs ;
  – communiquer une information élémentaire par l'intermédiaire du clavier ou de l'écran ;
  – coder l'information.

Du fait de la technologie, toutes les informations manipulées par un ordinateur sont codées en binaire (0 ou 1). Pour s'affranchir du langage machine binaire, on fait appel à un langage de programmation dit évolué, tel que les langages Pascal, C ou Java. Un tel programme se compose d'instructions définies par le langage, dont l'enchaînement réalise la solution du problème posé.

Pour traduire ce programme dans le langage binaire directement exécutable par l'ordinateur, nous devons utiliser un programme approprié, appelé compilateur ou interpréteur. Dans cet ouvrage, nous nous proposons d'étudier comment construire un programme en prenant comme support de langage, le **langage** et le **compilateur Java**.

Exercices

Apprendre à décomposer une tâche en sous-tâches distinctes

I.1 Écrivez la marche à suivre qui explique comment accrocher un tableau au centre d'un mur. Pour cela, vous devez :

a. Définir les objets nécessaires à la résolution du problème.

b. Établir la liste des opérations.

c. Ordonner cette liste.

✔ Plusieurs solutions sont possibles, mais chacune doit rester logique à l'égard des hypothèses prises en a. Par exemple, un clou et une perceuse ne vont pas ensemble.

Observer et comprendre la structure d'un programme Java

I.2 Observez le programme suivant :

```
public class Premier
{
```

```
  public static void main(String [] argument)
  {
   double a;
   System.out.print("Entrer une valeur : ") ;
   a = Lire.d() ;
   System.out.print(" Vous avez entre : " + a) ;
  }
}
```

a. Repérez les instructions définissant la fonction `main()` et celles délimitant la classe `Premier`.

b. Recherchez les instructions d'affichage.

c. Quel est le rôle de l'instruction `double a;` ?

d. Décrivez l'exécution de ce programme, en supposant que l'utilisateur entre au clavier la valeur `10`.

I.3 En suivant la structure ci-dessous et en vous aidant du programme donné à la section « Calcul de la circonférence d'un cercle », écrivez un programme qui calcule le périmètre d'un carré (rappel : périmètre = 4 × côté) :

```
public class ..........................// Donner un nom à la classe
{
  public static void main(String [] argument)
  {
   // Déclaration des variables représentant le périmètre et le côté
   ..............................................
   // Afficher le message "Valeur du côté : " à l'écran
   ..............................................
   // Lire au clavier une valeur
   // placer cette valeur dans la variable correspondante
   ..............................................
   // Calculer le périmètre du carré
   ..............................................
   // Afficher le résultat
   ..............................................
  }
}
```

Écrire un premier programme Java

I.4 En suivant la structure de l'exercice précédent, écrivez un programme qui calcule la surface d'un rectangle (rappel : surface = largeur × longueur).

En observant la formule :

a. Combien de variables faut-il déclarer pour exécuter le calcul ?

b. Combien de valeurs faut-il saisir au clavier ?

PARTIE 1

Les outils et techniques de base

1

Stocker une information

En décrivant, au chapitre introductif, « Naissance d’un programme », l’algorithme de confection d’un café chaud non sucré, nous avons constaté que la toute première étape pour construire une marche à suivre consistait à déterminer les objets utiles à la résolution du problème. En effet, pour faire du café, nous devons prendre le café, l’eau, le filtre, etc.

De la même façon, lorsqu’un développeur d’applications conçoit un programme, il doit non pas « prendre », au sens littéral du mot, les données numériques mais **définir** ces données ainsi que les objets nécessaires à la réalisation du programme. Cette définition consiste à nommer ces objets et à décrire leur contenu afin qu’ils puissent être stockés en mémoire.

C’est pourquoi nous étudions dans ce chapitre ce qu’est une variable et comment la définir (*« La notion de variable »*). Nous examinons ensuite, à la section « L’instruction d’affectation », comment placer une valeur dans une variable, par l’intermédiaire de l’instruction d’affectation. Enfin, nous analysons l’incidence du type des variables sur le résultat d’un calcul arithmétique (*« Les opérateurs arithmétiques »*).

Afin de clarifier les explications, vous trouverez tout au long du chapitre des exemples simples et concis. Ces exemples ne sont pas des programmes complets mais de simples extraits, qui éclairent un point précis du concept abordé. Vous trouverez en fin de chapitre (*« Calculer des statistiques sur des opérations bancaires »*), un programme entier qui aborde et résume toutes les notions rencontrées au fil de ce chapitre.

La notion de variable

Une variable permet la manipulation de données et de valeurs. Elle est caractérisée par les éléments suivants :

- Un **nom**, qui sert à repérer un emplacement en mémoire dans lequel une valeur est placée. Le choix du nom d’une variable est libre. Il existe cependant des contraintes, que nous présentons à la section « Les noms de variables ».

- Un **type**, qui détermine la façon dont est traduite la valeur en code binaire ainsi que la taille de l'emplacement mémoire. Nous examinons ce concept à la section « La notion de type ». Plusieurs types simples sont prédéfinis dans le langage Java, et nous en détaillons les caractéristiques à la section « Les types de base en Java ».

Les noms de variables

Le choix des noms de variables n'est pas limité. Il est toutefois recommandé d'utiliser des noms évocateurs. Par exemple, les noms des variables utilisées dans une application qui gère les codes-barres de produits vendus en magasin sont plus certainement « `article`, `prix`, `codebarre` » que « `xyz1`, `xyz2`, `xyz3` ». Les premiers, en effet, évoquent mieux l'information stockée que les seconds.

Les contraintes suivantes sont à respecter dans l'écriture des noms de variables :

- Le premier caractère d'une variable doit obligatoirement être différent d'un chiffre.
- Aucun espace ne peut figurer dans un nom.
- Les majuscules sont différentes des minuscules, et tout nom de variable possédant une majuscule est différent du même nom écrit en minuscule.
- Les caractères `&`, `~`, `"`, `#`, `'`, `{`, `}`, `(`, `)`, `[`, `]`, `-`, `|`, `` ` ``, `\`, `^`, `@`, `=`, `%`, `*`, `?`, `,` `:`, `/`, `§`, `!`, `<`, `>`, `£`, ainsi que `;` et `,` ne peuvent être utilisés dans l'écriture d'un nom de variable.
- Tout autre caractère peut être utilisé, y compris les caractères accentués, le caractère de soulignement (_) et les caractères `$`, `¤` et `µ`.
- Le nombre de lettres composant le nom d'une variable est indéfini. Néanmoins, l'objectif d'un nom de variable étant de renseigner le programmeur sur le contenu de la variable, il n'est pas courant de rencontrer des noms de variables de plus de trente lettres.

Exemples

Nom de variable autorisé	Nom de variable interdit	
`compte`	`pourquoi#pas`	caractère `#` interdit
`num_2` (" _ " et non pas " - ")	`2001espace`	pas de chiffre en début de variable
`undeux` (et non pas `un deux`)	`-plus`	caractère `-` interdit
`VALEUR_temporaire`	`@adresse`	caractère `@` interdit
`Val$solde`	`ah!ah!`	caractère `!` interdit

La notion de type

Un programme doit gérer des informations de nature diverse. Ainsi, les valeurs telles que `123` ou `2.4` sont de type numérique tandis que `Spinoza` est un mot composé de caractères. Si l'être humain sait, d'un simple coup d'œil, faire la distinction entre un nombre et un mot, l'ordinateur n'en est pas capable. Le programmeur doit donc « expliquer » à l'ordinateur la nature de chaque donnée. Cette explication passe par la notion de type.

Le type d'une valeur permet de différencier la nature de l'information stockée dans une variable.

À chaque type sont associés les éléments suivants :

- Un code spécifique permettant la traduction de l'information en binaire et réciproquement.
- Un ensemble d'opérations réalisables en fonction du type de variable utilisé. Par exemple, si la division est une opération cohérente pour deux valeurs numériques, elle ne l'est pas pour deux valeurs de type caractère.
- Un intervalle de valeurs possibles dépendant du codage utilisé. Par définition, à chaque type correspond un même nombre d'octets et, par conséquent, un nombre limité de valeurs différentes.

En effet, un octet, est un regroupement de 8 bits, sachant qu'un bit ne peut être qu'un `0` ou un `1`. Lorsqu'une donnée est codée sur 1 octet, elle peut prendre les valeurs `00000000` (8 zéros), ou encore `111111111` (8 un) et toutes les valeurs intermédiaires entre ces deux extrêmes (par exemple `10101010`, `11110000` ou `10010110`).

En fait, une donnée codée sur 8 bits peut, par le jeu des combinaisons de `0` et de `1`, prendre 2^8 valeurs différentes, soit 256 valeurs possibles comprises entre – 128 et 127. L'intervalle [– 128, 127] est en effet composé de 256 valeurs et possède autant de valeurs positives que négatives.

Pour représenter la valeur numérique 120, un codage sur 1 octet suffit, mais pour représenter la valeur 250, 1 octet ne suffit pas, et il est nécessaire d'utiliser un codage sur 2 octets.

Les types de base en Java

Chaque langage de programmation propose un ensemble de types de base permettant la manipulation de valeurs numériques entières, réelles ou caractères. Ces types sont :

- représentés par un mot-clé prédéfini par le langage.
- dits **simples**, car, à un instant donné, une variable de type simple ne peut contenir qu'une et une seule valeur.

À l'opposé, il existe des types appelés types **structurés** qui permettent le stockage, sous un même nom de variable, de plusieurs valeurs de même type ou non. Il s'agit des tableaux, des classes, des vecteurs ou encore des dictionnaires. Ces types structurés sont en général définis par le programmeur. Nous les étudions en détail dans la troisième partie de cet ouvrage, intitulée « Outils et techniques orientés objet ».

Pour sélectionner un type plutôt qu'un autre, le langage Java définit huit types simples, qui appartiennent, selon ce qu'ils représentent, à l'une ou l'autre des quatre catégories suivantes : logique, texte, entier, réel.

Catégorie logique

Il s'agit du type `boolean`. Les valeurs logiques ont deux états : « **`true`** » (vrai) ou « **`false`** » (faux). Elles ne peuvent prendre aucune autre valeur que ces deux états.

Catégorie caractère

Deux types définissent cette catégorie, le type `char` et le type `String`.

Le type `char` permet de représenter les caractères isolés, alors que le type `String` sert à décrire des séquences de caractères. En ce sens, il ne s'agit pas d'un type simple.

✔ Voir, au chapitre 7, « *Les classes et les objets* », la section « *La classe* `String`*, une approche vers la notion d'objet* ».

Pour décrire une variable de type `char`, l'ordinateur utilise un code sur 2 octets. De cette façon, il lui est possible d'utiliser jusqu'à 2^{16} caractères, soit 65 536 caractères différents. En réalité, Java utilise une table de correspondance, appelée jeu de caractères Unicode. Cette table associe un caractère à une valeur numérique.

Par exemple, dans la table Unicode, le caractère `A` majuscule a pour valeur décimale 65, et le caractère `a` minuscule la valeur décimale 97.

La table Unicode est organisée comme suit :

- Les 31 premiers caractères ne peuvent être affichés (tabulation, saut de ligne, bip sonore, etc.).
- Les caractères compris entre le 32e et le 127e correspondent aux caractères du code ASCII (*American Standard Code for Information Interchange*), qui était jusqu'à présent le code définissant tout caractère. Dans cet intervalle, tous les caractères de base sont définis, c'est-à-dire l'ensemble des lettres de l'alphabet, en minuscules et en majuscules, ainsi que les signes de ponctuation et les symboles mathématiques.
- Les caractères compris entre le 128e et le 256e caractères correspondent à des caractères spéciaux, comme les caractères accentués en minuscules et en majuscules et les caractères semi graphiques. Les codes de ces caractères font partie des extensions qui peuvent différer selon les pays ou les environnements de travail. Ces extensions sont définies à partir du jeu de caractères employé par votre environnement et diffèrent donc d'un type d'ordinateur à un autre.

✔ Pour connaître le code Unicode d'un caractère accentué sur votre ordinateur, reportez-vous à l'exemple de la section « *La boucle* `for` » du chapitre 4, « *Faire des répétitions* ».

- À partir du 257e caractère, il est possible de définir son propre jeu de caractères dans la table Unicode, de façon à représenter, par exemple, des caractères arabes, chinois ou japonais.

Catégorie entier

Cette catégorie contient quatre types distincts : `byte`, `short`, `int`, `long`. Chacun de ces types autorise la manipulation de valeurs numériques entières, positives ou négatives. Leur différence réside essentiellement dans le nombre d'octets utilisé pour coder le contenu de la variable.

Type	Nombre d'octets	Éventail de valeurs
byte	1 octet	de – 128 à 127
short	2 octets	de – 32 768 à 32 767
int	4 octets	de – 2 147 483 648 à 2 147 483 647
long	8 octets	de – 9 223 372 036 854 775 808 à 9 223 372 036 854 775 807

Dans certains cas, il est intéressant de représenter une valeur entière sous forme octale ou hexadécimale comme pour l'affichage des caractères de la table Unicode (*voir, au chapitre 2, « Communiquer une information », la section « Afficher les caractères accentués »*).

Valeur décimale	Valeur octale	Valeur hexadécimale
45	055	0x2d

Pour représenter un nombre sous forme octale, il est nécessaire de placer un zéro au début du nombre. Pour la représentation sous forme hexadécimale, les caractères `0x` doivent être placés en début de valeur.

Dans le langage Java, tous les types de la catégorie entier ont un signe (+ ou -).

Catégorie réel (flottant)

La catégorie réel permet l'emploi de nombres à virgule, appelés nombres réels ou encore flottants.

Deux types composent cette catégorie, le type `float` et le type `double`. Une expression numérique de cette catégorie peut s'écrire en notation décimale ou exponentielle.

- La notation décimale contient obligatoirement un point symbolisant le caractère « virgule » du chiffre à virgule. Les valeurs `67.3`, `-3.` ou `.64` sont des valeurs réelles utilisant la notation décimale.
- La notation exponentielle utilise la lettre `E` pour déterminer où se trouve la valeur de l'exposant (puissance de 10). Les valeurs 8.76E4 et 6.5E-12 sont des valeurs utilisant la notation exponentielle.

Dans les deux cas le nombre réel est suivi de la lettre `F` (pour `float`) ou `D` (pour `double`). Les caractères minuscules `f` ou `d` sont également autorisés. La distinction entre `float` et `double` s'effectue sur le nombre d'octets utilisé pour coder l'information. Il en résulte une précision plus ou moins grande suivant le type utilisé.

Type	Nombre d'octets	Éventail des valeurs
float	4 octets	de `1.40239846e-45F à 3.402823347e38F`
double	8 octets	de `4.94065645841246544e-324D` à `1.79769313486231570e308D`

Exemple

- La valeur 2.15F représente un simple flottant (type `float`).
- La valeur 1.35E22 représente aussi un flottant de grande taille.
- La valeur 6.76F est une valeur de type `float` de taille simple.
- La valeur 463.4E+234D correspond à un flottant de double précision (type `double`).

En langage Java, toute valeur numérique réelle est définie par défaut en double précision. Par conséquent, la lettre `d` (ou `D`) placée en fin de valeur n'est pas nécessaire. Par contre, dès que l'on utilise une variable `float`, la lettre `f` (ou `F`) est indispensable, sous peine d'erreur de compilation.

Comment choisir un type de variable plutôt qu'un autre ?

Sachant qu'une variable de type `int` (codée sur 4 octets) peut prendre toutes les valeurs de l'intervalle `[–2147483648/2147483647]` et donc prendre, en particulier, toutes les valeurs comprises entre `–32768` et `32767` (type `short`) ou même entre `–128` et `127` (type `byte`), posons-nous les questions suivantes :

- Pourquoi ne pas déclarer toutes les variables entières d'un programme en type `long` (le type `long` nous offrant le plus grand choix de valeurs entières) ?
- Pourquoi ne pas déclarer les variables réelles d'un programme en type `double` plutôt qu'en `float` ?

Pour répondre à ces questions, examinons le nombre d'octets utilisés par un programme de gestion de comptes bancaires. Pour simplifier, supposons que le programme garde en mémoire les 10 dernières opérations bancaires et le solde de chaque compte. Imaginons enfin que notre banque gère 50 000 comptes.

Pour stocker les 10 dernières opérations, nous devons déclarer 10 variables plus 1 pour le solde du compte, soit 11 variables. Les valeurs sont des montants en francs et centimes, donc des valeurs réelles.

- Si nous déclarons l'ensemble de ces variables en type `double` (8 octets), le programme utilise alors 50 000 × 11 × 8 octets, soit 44 000 000 octets, soit 4,4 mégaoctets de la mémoire de l'ordinateur.
- Si nous choisissons de prendre des variables de type `float` (ce qui reste cohérent, puisque les montants en francs n'ont pas besoin d'être d'une précision extrême), notre programme n'utilise plus que 2,2 mégaoctets, soit deux fois moins que précédemment.

Bien entendu, cet exemple simpliste n'a pour seul objectif que de montrer l'effet du choix du type de variable sur le taux d'occupation de la mémoire de l'ordinateur. Il existe, en réalité, un grand nombre de techniques pour optimiser la gestion de la mémoire de l'ordinateur.

Remarquons cependant que la première démarche pour gérer au mieux la mémoire de l'ordinateur consiste à bien choisir le type de ses variables. Si l'on sait que, par définition, une variable ne dépasse jamais, pour un programme donné, la valeur numérique 120, celle-ci doit être déclarée avec le type `byte`.

Déclarer une variable

La définition d'une variable dans un programme est réalisée par l'intermédiaire de l'instruction de **déclaration des variables**. Au cours de cette instruction, le programmeur donne le type et le nom de la variable. Pour déclarer une variable, il suffit d'écrire l'instruction selon la syntaxe suivante :

```
type nomdevariable ;
```

ou

```
type nomdevariable1, nomdevariable2 ;
```

où `type` correspond à l'un des mots-clés à choisir parmi ceux donnés aux sections précédentes (`boolean`, `char`, `String`, `byte`, `short`, `int`, `long`, `float` ou `double`). Si deux variables

de même type sont à déclarer, il n'est pas besoin de répéter le type, une virgule séparant les deux noms suffisant à les distinguer.

Pour expliquer à l'ordinateur que l'instruction de déclaration est terminée pour le type donné, un point virgule (;) est placé obligatoirement à la fin de la ligne d'instruction.

Exemple

```
float         f1, f2 ;         //Déclaration de deux variables de type float,
                               // une virgule sépare les deux noms de variables
long          CodeBar ;        //Déclaration d'une variable de type long
int           test ;           //Déclaration d'une variable de type int
char          choix, tmp ;     //Déclaration de deux variables de type char
boolean       OK ;             //Déclaration d'une variable de type boolean
```

Les instructions de déclaration peuvent être placées indifféremment au début ou en cours de programme. Une fois la variable déclarée, l'interpréteur Java réserve, au cours de l'exécution du programme, un emplacement mémoire correspondant en taille à celle demandée par le type. Il associe ensuite le nom de la variable à l'adresse de l'emplacement mémoire.

À cette étape du programme, remarquons que l'emplacement ainsi défini est vide. Si l'on souhaite afficher son contenu sans y avoir préalablement déposé de valeur, le compilateur émet le message d'erreur suivant : `Variable may not have been initialized`. Cette erreur indique que la variable dont on souhaite afficher le contenu n'a pas été initialisée. Comme l'interpréteur Java ne peut afficher un emplacement mémoire vide, l'exécution du programme n'est pas possible.

L'instruction d'affectation

Une fois la variable déclarée, il est nécessaire de stocker une valeur à l'emplacement mémoire désigné. Pour ce faire, nous utilisons l'instruction d'affectation, qui nous permet d'initialiser ou de modifier, en cours d'exécution du programme, le contenu de l'emplacement mémoire (le contenu d'une variable n'étant, par définition, pas constant).

Rôle et mécanisme de l'affectation

L'affectation est le mécanisme qui permet de placer une valeur dans un emplacement mémoire. Elle a pour forme :

```
Variable = Valeur ;
```

ou encore,

```
Variable = Expression mathématique ;
```

Le signe égal (=) symbolise le fait qu'une valeur est placée dans une variable. Pour éviter toute confusion sur ce signe mathématique bien connu, nous prendrons l'habitude de le traduire par les termes `prend la valeur`.

Examinons les exemples suivants, en supposant que les variables n et p soient déclarées de type entier :

```
n = 4 ;         //n prend la valeur 4
p = 5*n+1 ;     //calcule la valeur de l'expression mathématique soit 5*4+1
                //range la valeur obtenue dans la variable représentée par p.
```

L'instruction d'affectation s'effectue dans l'ordre suivant :

1. Calcule la valeur de l'expression figurant à droite du signe égal ;
2. Range le résultat obtenu dans la variable mentionnée à gauche du signe égal.

D'une manière générale, il est intéressant de remarquer que la variable placée à droite du signe = n'est jamais modifiée, alors que celle qui est à gauche l'est toujours. Comme une variable de type simple ne peut stocker qu'une seule valeur à la fois, si la variable située à gauche possède une valeur avant l'affectation, cette valeur est purement et simplement remplacée par la valeur située à droite du signe =.

Exemple

```
a = 1 ;
b = a + 3 ;
a = 3 ;
```

Lorsqu'on débute en programmation, une bonne méthode pour comprendre ce que réalise un programme consiste à écrire, pour chaque instruction exécutée, un état de toutes les variables déclarées. Il suffit pour cela de construire un tableau dont chaque colonne représente une variable déclarée dans le programme et chaque ligne une instruction de ce même programme. Soit, pour notre exemple :

instruction	a	b
a = 1	1	-
b = a + 3	1	4
a = 3	3	4

Le tableau est composé des deux colonnes a et b et des trois lignes associées aux instructions d'affectation du programme. Ce tableau montre que les instructions a = 1 et a = 3 font que la valeur initiale de a (1) est effacée et écrasée par la valeur 3.

Déclaration et affectation

Comme nous l'avons vu à la section « Déclarer une variable », la déclaration est utilisée pour réserver un emplacement mémoire. Une fois réservé, l'emplacement reste vide jusqu'à ce qu'une valeur y soit placée par l'intermédiaire de l'affectation.

Il est cependant risqué de déclarer une variable sans lui donner de valeur initiale. En effet, le compilateur Java vérifie strictement si toutes les variables contiennent une valeur ou non. Une erreur de compilation est détectée dès qu'une seule variable ne contient pas de valeur à un moment donné du programme.

Initialiser une variable

Pour éviter toute erreur de compilation, une bonne habitude consiste à initialiser toutes les variables au moment de leur déclaration, en procédant de la façon suivante :

```
float      f1 = 0.0f, f2 = 1.2f ;  // Initialisation de deux float
long       CodeBar = 123456789 ;   // Initialisation d'un long
int        test = 0 ;              // Initialisation d'une variable de type int
boolean    OK = true ;             // Initialisation d'un boolean
```

De cette façon, les variables `f1`, `f2`, `CodeBar` et `OK` sont déclarées. Le compilateur réserve un emplacement mémoire pour chacune d'entre elles. Grâce au signe d'affectation, le compilateur place dans chacun des emplacements mémoire respectifs les valeurs données.

Initialiser une variable de type `char`

Les variables de type `char` s'initialisent d'une façon particulière. Supposons que l'on souhaite déclarer et placer le caractère `n` dans une variable `choix` de type `char`. Pour cela, écrivons l'instruction de déclaration et d'initialisation suivante :

```
char choix = n ;
```

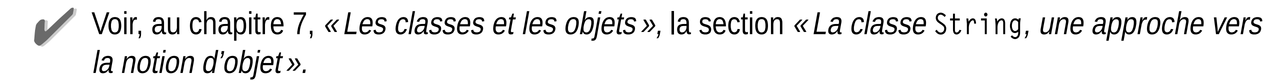
Pour le compilateur, cette instruction est problématique, car il considère `n` non pas comme le « caractère `n` » mais comme une variable appelée `n`.

Pour lever cette ambiguïté, nous devons entourer le caractère `n` d'apostrophes, de la façon suivante :

```
char choix = 'n' ;
```

Ainsi, des données telles que `'a'`, `'*'`, `'$'`, `'3'`, `':'` ou `'?'` sont considérées comme des caractères.

Par contre `c = 'ab'` ne peut s'écrire, car `'ab'` n'est pas un caractère mais un mot de deux caractères. Nous devons, dans ce cas, utiliser une variable de type `String`.


Voir, au chapitre 7, *« Les classes et les objets »*, la section *« La classe* `String`*, une approche vers la notion d'objet »*.

Quelques confusions à éviter

Le symbole de l'affectation est le signe égal (`=`). Ce signe, très largement utilisé dans l'écriture d'équations mathématiques, est source de confusion lorsqu'il est employé à contre-sens.

Pour mieux nous faire comprendre, étudions trois cas :

1. a = a + 1 ;

   Si cette expression est impossible à écrire d'un point de vue mathématique, elle est très largement utilisée dans le langage informatique. Elle signifie :

   – calculer l'expression `a + 1` ;

– ranger le résultat dans `a`.

Ce qui revient à augmenter de 1 la valeur de a.

2. a + 5 = 3 ;

   Cette expression n’a aucun sens d’un point de vue informatique. Il n’est pas possible de placer une valeur à l’intérieur d’une expression mathématique, puisque aucun emplacement mémoire n’est attribué à une expression mathématique.

3. a = b ; et b = a ;

   À l’inverse de l’écriture mathématique, ces deux instructions ne sont pas équivalentes. La première place le contenu de `b` dans `a`, tandis que la seconde place le contenu de `a` dans `b`.

Échanger les valeurs de deux variables

Nous souhaitons échanger les valeurs de deux variables de même type, appelées `a` et `b` ; c’est-à-dire que nous voulons que `a` prenne la valeur de `b` et que `b` prenne celle de `a`. La pratique courante de l’écriture des expressions mathématiques fait que, dans un premier temps, nous écrivions les instructions suivantes :

```
a = b ;
b = a ;
```

Vérifions sur un exemple si l’exécution de ces deux instructions échange les valeurs de `a` et de `b`. Pour cela, supposons que les variables `a` et `b` contiennent initialement respectivement `2` et `8`.

	a	b
`valeur initiale`	2	8
`a = b`	8	8
`b = a`	8	8

Du fait du mécanisme de l’affectation, la première instruction `a = b` détruit la valeur de `a` en plaçant la valeur de `b` dans la case mémoire `a`. Lorsque la seconde instruction `b = a` est réalisée, la valeur placée dans la variable `b` est celle contenue à cet instant dans la variable `a`, c’est-à-dire la valeur de `b`. Il n’y a donc pas échange, car la valeur de `a` a disparu par écrasement lors de l’exécution de la première instruction.

Une solution consiste à utiliser une variable supplémentaire, destinée à contenir temporairement une copie de la valeur de `a`, avant que cette dernière soit écrasée par la valeur de `b`. Pour évoquer le caractère temporaire de la copie, nous appellerons cette nouvelle variable `tmp` (nous aurions pu choisir tout aussi bien `tempo` ou `ttt`). Voici le déroulement des opérations :

```
tmp = a ;
a = b ;
b = tmp ;
```

Vérifions qu'il y a réellement échange, en supposant que nos variables `a` et `b` contiennent initialement respectivement `2` et `8`.

	a	b	tmp
`valeur initiale`	2	8	_
`tmp = a`	2	8	2
`a = b`	8	8	2
`b = tmp`	8	2	2

À la lecture de ce tableau, nous constatons qu'il y a bien échange des valeurs entre `a` et `b`. La valeur de `a` est copiée dans un premier temps dans la variable `tmp`. La valeur de `a` peut dès lors être effacée par celle de `b`. Pour finir, grâce à la variable `tmp` la variable `b` récupère l'ancienne valeur de `a`.

 Une autre solution vous est proposée dans la feuille d'exercices placée à la fin du chapitre.

Les opérateurs arithmétiques

Écrire un programme n'est pas uniquement échanger des valeurs, mais c'est aussi calculer des équations mathématiques plus ou moins complexes. Pour exprimer une opération, le langage Java utilise des caractères qui symbolisent les opérateurs arithmétiques.

Symbole	Opération
+	addition
-	soustraction
*	multiplication
/	division
%	modulo

Exemple

Soient `a`, `b`, `c` trois variables de même type.

- L'opération d'addition s'écrit : `a = b + 4`.
- L'opération de soustraction s'écrit : `a = b – 5`.
- L'opération de division s'écrit : `a = b / 2` et non pas $a = \frac{b}{2}$.
- L'opération de multiplication s'écrit : `a = b * 4`.
  et non pas `a = 4b` ou `a = a x b`.
- L'opération de modulo s'écrit : `a = b % 3`.

Le modulo d'une valeur correspond au reste de la division entière. Ainsi : `5 % 2 = 1`

Il s'agit de calculer la division en s'arrêtant dès que le valeur du reste devient inférieure au diviseur, de façon à trouver un résultat en nombre entier. L'opérateur `%` n'existe pas pour les réels, pour lesquels la notion de division entière n'existe pas.

L'ensemble de ces opérateurs est utilisé pour calculer des expressions mathématiques courantes. Le résultat de ces expressions n'est cependant pas toujours celui auquel on s'attend. Trois phénomènes ont une influence non négligeable sur la valeur du résultat d'un calcul. Ce sont :

- La priorité des opérateurs entre eux ;
- Le type d'une expression mathématique ;
- La transformation de types.

La priorité des opérateurs entre eux

Lorsqu'une expression arithmétique est composée de plusieurs opérations, l'ordinateur doit pouvoir déterminer quel est l'ordre des opérations à effectuer. Le calcul de l'expression `a - b / c * d` peut signifier *a priori* :

- calculer la soustraction puis la division et pour finir la multiplication, soit le calcul : `((a - b) / c) * d` ;
- calculer la multiplication puis la division et pour finir la soustraction, c'est-à-dire l'expression : `a - (b / (c * d))`.

Afin d'éviter toute ambiguïté, il existe des règles de priorité entre les opérateurs, règles basées sur la définition de deux groupes d'opérateurs.

Groupe 1	Groupe 2
+ -	* / %

Les groupes étant ainsi définis, les opérations sont réalisées sachant que :

- Dans un même groupe, l'opération se fait dans l'ordre d'apparition des opérateurs (sens de lecture).
- Le deuxième groupe a priorité sur le premier.

L'expression `a - b / c * d` est calculée de la façon suivante :

Priorité	Opérateur	
Groupe 2	/	Le groupe 2 a priorité sur le groupe 1, et la division apparaît dans le sens de la lecture avant la multiplication.
Groupe 2	*	Le groupe 2 a priorité sur le groupe 1, et la multiplication suit la division.
Groupe 1	-	La soustraction est la dernière opération à exécuter, car elle est du groupe 1.

Cela signifie que l'expression est calculée de la façon suivante :

```
a - (b / c * d)
```

Remarquons que les parenthèses permettent de modifier les règles de priorité en forçant le calcul préalable de l'expression qui se trouve à l'intérieur des parenthèses. Elles offrent en outre une meilleure lisibilité de l'expression.

Le type d'une expression mathématique

Le résultat d'une expression mathématique peut être déterminé à partir du type de variables (termes) qui composent l'expression.

Terme	Opération	Terme	Résultat
Entier	+ - * / %	Entier	Entier
Réel	+ - * /	Réel	Réel

De ce fait, pour un même calcul, le résultat diffère selon qu'il est effectué à l'aide de variables de type réel ou de type entier.

Exemple : diviser deux entiers

```
int x = 5 , y = 2, z ;
z = x / y ;
```

	x	y	z
valeur initiale	5	2	-
z = x / y	5	2	2

Ici, toutes les variables déclarées sont de type entier. Par conséquent, l'opération effectuée a pour résultat une valeur entière, même si l'opération demandée n'a pas forcément un résultat entier. Soit, pour notre exemple, 2 et non 2.5. Cela ne correspond pas toujours au résultat attendu par le programmeur débutant.

Exemple : diviser deux réels

```
double x = 5 , y = 2, z ;
z = x / y ;
```

	x	y	z
valeur initiale	5	2	-
z = x / y	5	2	2.5

Ici, toutes les variables déclarées sont de type réel. Par conséquent, l'opération effectuée donne un résultat de type réel. Ce résultat correspond à la valeur généralement attendue de ce type d'opération.

La transformation de types

Les termes d'une opération ne sont pas nécessairement tous du même type. Pour écrire une opération, toutes les combinaisons entre les différentes catégories de types peuvent se présenter.

Terme	Opération	Terme	Résultat
byte	+ - * /	int	int
int	+ - * /	double	double

L'ordinateur ne sait calculer une expression mathématique que lorsque toutes les variables de l'expression sont du même type. En effet, les opérateurs arithmétiques ne sont définis que pour des variables de type identique.

Lorsque tel n'est pas le cas, c'est-à-dire si l'expression est mixte, l'ordinateur doit transformer le type de certaines variables pour que tous les membres de l'expression deviennent de même type.

Cette transformation, appelée **conversion d'ajustement de type**, se réalise suivant une hiérarchie bien déterminée, qui permet de ne pas perdre d'information. On dit que le compilateur respecte l'intégralité des données.

La conversion d'un nombre réel en nombre entier, par exemple, ne peut se réaliser qu'en supprimant les nombres situés après la virgule et en ne gardant que la partie entière du nombre. Une telle conversion ne garantit pas l'intégralité des données car il y a perte de données.

C'est pourquoi, du fait du codage des données et du nombre d'octets utilisé pour ce codage, le compilateur effectue automatiquement la conversion des données selon l'ordre suivant :

```
byte -> short -> int -> long -> float -> double
```

De cette façon, il est toujours possible de convertir un `byte` en `long` ou un `int` en `float`. Par contre, il est impossible de transformer un `float` en `short` sans perte d'information.

Exemple

```
int a = 4, result_int ;
float x = 2.0f, result_float ;
result_float = a / x ;
result_int = a / x ;
```

	a	x	result_float	result_int
a = 4	4	_	_	_
x = 2.0f	4	2.0f	_	_
result_float = a/x	4	2.0f	2.0f	_
result_int = a/x	4	2.0f	_	Impossible dès la compilation

La troisième instruction montre que le calcul d'une opération dont les termes sont de type `int` et `float` donne pour résultat un `float`. La dernière instruction révèle que, si le résultat d'une opération est de type `float`, il n'est pas possible de le stocker dans une variable de type `int`. En effet, la division d'un entier par un réel est une opération toujours possible à réaliser (le résultat est de type réel), mais l'affectation directe de ce résultat dans une variable entière est impossible du fait que la conversion entraîne une perte d'information.

Une telle instruction provoque à la compilation une erreur dont le message est : `Incompatible type for =. Explicit cast needed to convert float to int.` Cela signifie : « Type incompatible de part et d'autre du signe =. Pour convertir un `float` en `int`, il est nécessaire de le formuler explicitement par l'intermédiaire d'un cast. »

Le cast

La conversion avec perte d'information est autorisée dans certain cas grâce au mécanisme du *cast*. Il peut être utile de transformer un nombre réel en entier, par exemple pour calculer sa partie entière. Pour cela, le compilateur demande de convertir explicitement les termes de l'opération dans le type souhaité en plaçant devant la variable ou l'opération le type de conversion désiré. Ainsi, pour transformer un `float` en `int`, il suffit de placer le terme `(int)` devant la variable ou l'opération de type `float`.

Exemple

```
int a = 5, result ;
float x = 2.0f ;
result = (int) a / x ;
```

	a	x	result
a = 5	5	_	_
x = 2.0f	5	2.0f	_
result = (int) a / x	5	2.0f	2

La dernière instruction montre que la conversion `float` vers `int` est autorisée malgré la perte d'information (le chiffre 5 placé après la virgule disparaît). Cette conversion n'est possible que si elle est précisément indiquée au compilateur.

Calculer des statistiques sur des opérations bancaires

Pour résumer en pratique l'ensemble des notions abordées dans ce chapitre, nous allons écrire un programme, dont le sujet se rapporte au thème du projet énoncé à la fin du chapitre introductif, « Naissance d'un programme ».

Cahier des charges

L'objectif de ce programme est d'établir des statistiques sur l'utilisation des différents modes de paiement effectués sur un compte bancaire. Nous supposons que les moyens techniques pour débiter un compte sont au nombre de trois : la Carte Bleue, le chéquier et le virement. Pour évaluer le taux d'utilisation de ces trois moyens de paiement, nous devons calculer le pourcentage d'utilisation de chaque technique par rapport aux autres. Par exemple, pour connaître le pourcentage d'utilisation de la Carte Bleue, nous utilisons le calcul suivant :

Nombre de paiements par Carte Bleue / Nombre total de paiements * 100

Liste des opérations

Partant du principe de décomposition d'un problème en sous-tâches plus simples à réaliser, distinguons, pour résoudre la question, les quatre actions suivantes :

1. Déterminer le nombre de débits par Carte Bleue, chèque et virement. Comme il s'agit du premier programme concernant ce thème, nous n'avons pas encore saisi de valeur, ni de ligne comptable. C'est pourquoi nous demandons à l'utilisateur de communiquer au programme ces trois informations, par l'intermédiaire du clavier.
2. Calculer le nombre total de paiements effectués.
3. Calculer le pourcentage d'utilisation de la Carte Bleue, du chéquier et du virement.
4. Afficher l'ensemble des résultats.

Dans un premier temps, nous traiterons séparément chacun de ces points afin de les analyser entièrement. Pour finir, nous écrirons le programme dans son intégralité, en regroupant chacun des points étudiés.

1. Il s'agit d'écrire les instructions qui permettent à l'utilisateur de communiquer des informations à l'ordinateur à l'aide du clavier. Nous avons vu, au chapitre introductif, un exemple de saisie d'une valeur au clavier (*voir « Calcul de la circonférence d'un cercle »*). Cette opération se réalise en deux temps : d'abord l'affichage à l'écran d'un message informant l'utilisateur d'une demande de saisie de valeur, puis la saisie effective de l'information. Pour notre problème, ces deux points se traduisent de la façon suivante :

```
System.out.print(" Nombre de paiement par Carte Bleue ") ;
nbCB = Lire.i() ;
System.out.print(" Nombre de cheques émis ") ;
nbCheque = Lire.i() ;
System.out.print(" Nombre de virements automatiques ") ;
nbVirement = Lire.i() ;
```

Chaque appel de la fonction `System.out.print()` affiche à l'écran le message placé entre `guillemets`. Trois messages sont affichés, chacun indiquant respectivement à quel mode de paiement est associée la valeur saisie par l'utilisateur.

Les valeurs à saisir correspondent aux nombres de débits dans chaque mode de paiement. Ces valeurs sont de type entier. La fonction `Lire.i()` donne l'ordre à l'ordinateur d'attendre la saisie d'une valeur entière. La saisie est effective lorsque l'utilisateur valide sa réponse en appuyant sur la touche « `Entrée` » du clavier. Trois valeurs sont à saisir, et il est nécessaire d'appeler trois fois la fonction `Lire.i()`.

✔ Pour plus d'informations sur la fonction `Lire.i()`, voir le chapitre 2, *« Communiquer une information »*.

Une fois saisie, chaque valeur doit être stockée dans un emplacement mémoire distinct. Ces emplacements mémoire correspondent aux trois variables `nbCB`, `nbCheque` et `nbVirement` et sont déclarés en début de programme grâce à l'instruction :

```
int nbCB = 0, nbCheque = 0, nbVirement = 0 ;
```

2. Pour calculer le nombre total de paiements effectués, il suffit de faire la somme de toutes les opérations de débit pour tous les types de paiement, soit l'instruction :

```
nbDebit = nbCB + nbCheque + nbVirement ;
```

La variable `nbDebit` permet la mémorisation du nombre total d'opérations effectuées, quel que soit le mode de paiement. Elle doit être déclarée en même temps que les autres variables du même type :

```
int nbCB = 0, nbCheque = 0, nbVirement = 0, nbDebit = 0 ;
```

3. Pour calculer le pourcentage d'utilisation de la Carte Bleue, du chéquier et du virement, nous allons d'abord étudier le mode Carte Bleue puis appliquer cette analyse aux autres modes de paiement. Rappelons que la formule du calcul de pourcentage pour la Carte Bleue est :

Nombre de paiements par Carte Bleue / Nombre total de paiements * 100

Soit, en utilisant les variables déclarées au point 1 : `nbCB / nbDebit * 100`.

Examinons sur un exemple numérique le résultat d'un tel calcul. Supposons pour cela que nous ayons effectué 10 retraits Carte Bleue sur un total de 40 retraits. Nous obtenons le calcul suivant : `10 / 40 * 100`. Soit `0 * 100`, c'est-à-dire `0`. La division est la première opération exécutée parce qu'elle est du même groupe que la multiplication et qu'elle apparaît en premier dans l'opération. De surcroît, les valeurs étant de type entier, la division a pour résultat un nombre entier. Ici `10/40` a pour résultat `0`.

Pour corriger cette erreur de calcul, l'idée est de réaliser une division sur des valeurs réelles et non sur des entiers. Pour cela, nous utilisons le mécanisme du cast, qui, placé devant la variable `nbCB`, transforme cette dernière en variable de type réel et permet la division en réel. Pour stocker le résultat de cette opération, nous déclarons une variable de type `float`, nommée `prctCB`.

L'instruction :

```
prctCB = (float) nbCB / nbDebit * 100 ;
```

permet de trouver un résultat cohérent. Vérifions cela sur un exemple numérique. Supposons que nous ayons effectué 10 débits par Carte Bleue sur un total de 20 retraits. Grâce au cast, la valeur `10` correspondant à `nbCB` est transformée en `10.0`. La division par 20 a donc un résultat réel égal à 0.5. Le taux d'utilisation de la Carte Bleue est donc de `0.5 * 100`, soit 50 %.

Pour établir le pourcentage relatif aux modes chéquier et virement, il suffit d'appliquer le même calcul, en utilisant des variables appropriées aux deux autres moyens de paiement. En nommant `prctCh` et `prctVi` les variables associées aux modes de paiement par chèque et par virement automatique, le taux d'utilisation pour chacun de ces modes s'écrit :

```
prctCh = (float) nbCheque / nbDebit * 100 ;
prctVi = (float) nbVirement / nbDebit * 100 ;
```

4. L'affichage des résultats s'effectue par l'intermédiaire de la fonction `System.out.print()`. Les valeurs calculées sont commentées de la façon suivante :

```
System.out.println(" Vous avez emis " + nbDebit + " ordres de debit ") ;
System.out.println(" dont " + prctCB + " % par Carte Bleue ") ;
System.out.println("      " + prctCh + " % par cheque ") ;
System.out.println("      " + prctVi + " % par virement ") ;
```

Le programme final s'écrit en regroupant l'ensemble des instructions définies précédemment et en les insérant dans une classe à l'intérieur de la fonction `main()`.

Le code source complet

```
public class Statistique
{
  public static void main (String [] arg)
  {
   int nbCB = 0, nbCheque = 0, nbVirement = 0, nbDebit = 0 ;
   float prctCB, prctCh, prctVi ;
   System.out.print(" Nombre de paiements par Carte Bleue : ") ;
   nbCB = Lire.i() ;
   System.out.print(" Nombre de cheques emis : ") ;
   nbCheque = Lire.i() ;
   System.out.print(" Nombre de virements automatiques : ") ;
   nbVirement = Lire.i() ;

   nbDebit = nbCB + nbCheque + nbVirement;

   prctCB = (float) nbCB / nbDebit * 100 ;
   prctCh = (float) nbCheque / nbDebit * 100 ;
   prctVi = (float) nbVirement / nbDebit * 100 ;

  System.out.println("Vous avez emis " + nbDebit + " ordres de debit") ;
  System.out.println("dont " + prctCB + " % par Carte Bleue") ;
  System.out.println("     " + prctCh + " % par cheque") ;
  System.out.println("     " + prctVi + " % par virement") ;
  }
}
```

Résultat de l'exécution

À l'exécution de ce programme, nous avons à l'écran l'affichage suivant (les caractères grisés sont des valeurs choisies par l'utilisateur) :

```
Nombre de paiements par Carte Bleue : 5
Nombre de cheques emis : 10
Nombre de virements automatiques : 5
Vous avez emis 20 ordres de debit
dont 25.0 % par Carte Bleue
     50.0 % par cheque
     25.0 % par virement
```

Résumé

Une **variable** est caractérisée par un **nom** et un **type**. Le nom sert à repérer un emplacement mémoire. Le type détermine la taille de cet emplacement, ainsi que la manière dont l'information est codée, les opérations autorisées et l'intervalle des valeurs représentables.

Il existe plusieurs types simples, dont les plus utilisés sont les suivants :

- **`int`**. Présente les entiers variant, pour le langage Java, entre – 2 147 483 648 et 2 147 483 647.
- **`double`**. Décrit de manière approchée les nombres réels dont la valeur absolue est grande. Les variables de type `double` se notent soit sous forme décimale (67.7, –9.2, 0.48 ou .22), soit sous forme exponentielle 3.14E4, .325707e2, –45.567E-5.
- **`char`**. Désigne les caractères. Les valeurs de type caractère se notent en plaçant entre apostrophes le caractère lui-même.

L'instruction d'**affectation** permet de placer une valeur dans une variable. Elle est de la forme : `variable = expression;`.

Elle calcule d'abord la valeur de l'expression mentionnée à droite du signe `=`, puis elle l'affecte à la variable placée à gauche du signe.

Il est conseillé d'attribuer une valeur initiale à une variable au moment de sa déclaration. Par exemple `int i = 6;` ou `char c = 'n';`.

Pour calculer des expressions mathématiques, il existe cinq **opérateurs** arithmétiques : `+ - * / %`.

Ces opérateurs sont utilisés respectivement pour l'addition, la soustraction, la multiplication, la division et le modulo (reste de la division entière). Les expressions arithmétiques sont calculées à partir des règles suivantes :

- Entier `+ - * / %` entier donne un entier.
- Réel `+ - * /` réel donne un réel.
- Les opérations mixtes du type :

```
entier + - * / réel ou réel + - * / entier
```

donnent un résultat dans la mesure où la valeur résultante n'est pas dénaturée par la conversion des types. Les conversions sont effectuées automatiquement dans le sens suivant :

```
byte -> short -> int -> long -> float -> double
```

Un `int` peut donc être transformé en un `double`. L'inverse n'est possible que lorsque le mode de conversion est explicitement décrit dans l'expression, comme dans `n = (int) x`, où `n` est de type `int` et `x` de type `double`.

L'information ainsi transformée est tronquée pour être codée sur moins d'octets.

- Il existe des règles de **priorit**é entre les opérateurs. Pour cela, deux groupes d'opérateurs sont définis.

Groupe 1	Groupe 2
+ -	* / %

Dans un même groupe, l'opération se fait dans l'ordre d'apparition des opérateurs.

Le second groupe a priorité sur le premier.

Les parenthèses permettent la modification des priorités.

Exercices

Repérer les instructions de déclaration, observer la syntaxe d'une instruction

1.1 Observez ce qui suit, et indiquez ce qui est ou n'est pas une déclaration et ce qui est ou n'est pas valide :

```
a.  int i, j, valeur ;
b.  limite - j = 1024 ;
c.  val = valeur / 16 ;
d.  char char ;
e.  j + 1 ;
f.  int X ;
g.  float A ;
h.  A = X / 2 ;
i.  X = A / 2 ;
j.  X = X / 2 ;
```

Comprendre le mécanisme de l'affectation

1.2 Quelles sont les valeurs des variables `A`, `B`, `C` après l'exécution de chacun des extraits de programme suivants :

a.	b.

`float A = 3.5f ;`
`float B = 1.5f ;`
`float C ;`
`C = A + B ;`
`B = A + C ;`
`A = B ;`
	`double A = 0.1 ;`
	`double B = 1.1 ;`
	`double C, D ;`
	`B = A ;`
	`C = B ;`
	`D = C ;`
	`A = D ;`

1.3 Quelles sont les valeurs des variables `a`, `b` et `c`, `valeur`, `x`, `y` et `z`, après l'exécution de chacune des instructions suivantes :

a.	b.	c.
`int a = 5, b ;`	`int valeur = 2 ;`	`int x = 2, y = 10, z ;`
`b = a + 4 ;`	`valeur = valeur + 1 ;`	`z = x + y ;`
`a = a + 1 ;`	`valeur = valeur * 2 ;`	`x = 5 ;`
`b = a - 4 ;`	`valeur = valeur % 5 ;`	`z = z - x ;`

Comprendre le mécanisme d'échange de valeurs

1.4 Dans chacun des cas, quelles sont les valeurs des variables `a` et `b` après l'exécution de chacune des instructions suivantes :

1.	2.
`int a = 5`	`int a = 5`
`int b = 7`	`int b = 7`
`a = b`	`b = a`
`b = a`	`a = b`

1.5 Laquelle des options suivantes permet d'échanger les valeurs des deux variables `a` et `b` ?

```
a = b ; b = a ;
t = a ; a = b ; b = t ;
t = a ; b = a ; t = b ;
```

1.6 Soit trois variables `a`, `b` et `c` (entières). Écrivez les instructions permutant les valeurs, de sorte que la valeur de `a` passe dans `b`, celle de `b` dans `c` et celle de `c` dans `a`. N'utilisez qu'une (et une seule) variable entière supplémentaire, nommée `tmp`.

1.7 Quel est l'effet des instructions suivantes sur les variables `a` et `b` (pour vous aider, initialisez `a` à 2 et `b` à 5) :

```
a = a + b ;
b = a - b ;
a = a - b ;
```

Calculer des expressions mixtes

1.8 Donnez les valeurs des expressions suivantes, sachant que `i` et `j` sont de type `int` et `x` et `y` de type `double` (`x = 2.0`, `y = 3.0`) :

```
a.          i = 100 / 6 ;
b.          j = 100 % 6 ;
c.          i = 5 % 8
d.          (3 * i - 2 * j) / (2 * x - y)
e.          2 * ((i / 5) + (4 * (j - 3)) % (i + j - 2))
f.          (i - 3 * j) / (x + 2 * y) / (i - j)
```

1.9 Donnez le type et la valeur des expressions suivantes, sachant que `n`, `p`, `r`, `s` et `t` sont de type `int` (`n = 10`, `p = 7`, `r = 8`, `s = 7`, `t = 21`) et que `x` est de type `float` `(x = 2.0f )` :

a.	b.
`x + n % p`	`r + t / s`
`x + n / p`	`( r + t ) / s`
`(x + n) / p`	`r + t % s`
`5. * n`	`(r + t) % s`
`(n + 1) / n`	`r + s / r + s`
`(n + 1.0) / n`	`(r + s) / ( r + s)`
`r + s / t`	`r + s % t`

Comprendre le mécanisme du cast

1.10 Soit les déclarations suivantes :

```
int valeur = 7, chiffre = 2, i1, i2 ;
float f1, f2 ;
```

Quelles sont les valeurs attribuées à `i1`, `i2`, `f1` et `f2` après le calcul de :

```
i1 = valeur / chiffre ;
i2 = chiffre / valeur ;
f1 = (float) (valeur / chiffre) ;
f2 = (float) (valeur / chiffre) + 0.5f ;
i1 = (int) f1 ;
i2 = (int) f2 ;
f1 = (float) valeur / (float) chiffre ;
f2 = (float) valeur / (float) chiffre + 0.5f ;
i1 = (int) f1 ;
i2 = (int) f2 ;
```

Le projet « Gestion d'un compte bancaire »

Déterminer les variables nécessaires au programme

Le programme de gestion d'un compte bancaire ne peut s'écrire et s'exécuter sans aucune variable. Pour pouvoir définir toutes les variables nécessaires à la bonne marche du programme, nous devons examiner attentivement le cahier des charges décrit au chapitre introductif, « Naissance d'un programme ».

La section « Les objets manipulés » nous donne une première idée des variables à déclarer. Toutes les données relatives au compte bancaire y sont décrites.

Un compte bancaire est défini par un ensemble de données :

- un numéro de compte ;
- un type de compte (courant, épargne, joint, etc.) ;

- des lignes comptables possédant chacune une valeur, une date, un thème et un moyen de paiement.

Ces données peuvent être représentées de la façon suivante :

Données	Exemple	Type de l'objet
Numéro du compte	4010.205.530	Suite de caractères
Type du compte	Courant	Suite de caractères
Valeur	–1520.30	Numérique
Date	04 03 1978	Date
Thème	Loyer	Suite de caractères
Moyen de paiement	CB	Suite de caractères

Compte tenu de ces informations, donnez un nom et un type Java pour chaque donnée définie ci-dessus.

Remarquons que le type qui représente les suites de caractères (`String`) n'a pas encore été étudié, ni toutes ses fonctionnalités. Il est possible de transformer pour l'instant les données `Type du compte`, `Thème` et `Moyen de paiement` en caractères simples. Par exemple, le caractère `C` caractérise le type du compte `Courant`, le caractère `J` le compte `Joint` et le caractère `E` le compte `Epargne`.

De la même façon, la donnée `Numéro du compte` peut être transformée dans un premier temps en type `long`.

2

Communiquer une information

Un programme n'a d'intérêt que s'il produit un résultat. Pour communiquer ce résultat, l'ordinateur utilise l'écran. Cette action, qui consiste à afficher un message, est appelée opération de **sortie**, ou d'écriture, de données.

Parallèlement, un programme ne produit de résultats que si l'utilisateur lui fournit au préalable des informations. Ces informations, ou données, sont transmises au programme le plus souvent par l'intermédiaire d'un clavier. Dans le jargon informatique, cette opération est appelée opération de saisie, d'**entrée** ou encore de lecture de données.

Dans ce chapitre, nous commençons par étudier les fonctionnalités proposées par le langage Java pour gérer les opérations d'entrée-sortie (*« La bibliothèque* `System` *»).*

À la section « L'affichage de données », nous examinons ensuite comment afficher à l'écran des messages et des données. Enfin, à la section « La saisie de données », nous proposons une technique de saisie de valeurs au clavier.

La bibliothèque System

Nous l'avons vu dans les exemples des chapitres précédents, l'affichage de valeurs ou de texte est réalisé par l'utilisation d'une fonction prédéfinie du langage Java. Cette fonction a pour nom d'appel `System.out.print()`.

Pourquoi un nom si complexe, pour réaliser une action aussi « simple » que l'affichage de données ?

Le langage Java est accompagné d'un ensemble de bibliothèques de programmes préécrits, qui épargnent au programmeur d'avoir à réécrire ce qui a déjà été fait depuis les débuts de l'ère informatique. Ces bibliothèques portent chacune un nom qui renseigne

sur leur fonctionnalité. Ainsi, la bibliothèque où se trouve l'ensemble des fonctions de calcul mathématique s'appelle `Math`, et celle relative à la gestion des éléments de bas niveau (écran, clavier, etc.) impliquant le système de l'ordinateur s'appelle `System`.

La gestion de l'affichage d'un message à l'écran ou la saisie de valeurs au clavier font partie des fonctions impliquant le système de l'ordinateur. C'est pourquoi le nom d'appel de telles fonctions a pour premier terme `System`.

Les opérations d'entrée ou de sortie de données impliquent le système de l'ordinateur mais sont en rapport inverse l'une de l'autre. Pour dissocier ces opérations, la librairie `System` est composée de deux sous-ensembles, `in` et `out`. L'affichage est une opération de sortie et fait donc partie des éléments `out` de la classe `System`. Le point (`.`) qui relie le mot `System` à `out` permet d'expliquer à l'ordinateur que l'on souhaite accéder au sous-ensemble `out` de la librairie `System` plutôt qu'au sous-ensemble `in`. Pour finir, nous faisons appel, dans le sous-ensemble `out`, à la fonction `print`, qui affiche un message à l'écran. Le nom de la fonction `print` signifie imprimer, car, au tout début de l'informatique, les ordinateurs n'avaient pas d'écran, et les résultats d'un calcul étaient imprimés sur papier ou sur carte informatique.

La notation point (`.`) est une écriture courante en programmation objet. Comme nous le verrons au chapitre 7, « Les classes et les objets », elle offre le moyen d'accéder à des programmes ou à des données spécifiques.

Notons que, dans la classe `System`, se trouve aussi le sous-ensemble `err`, qui permet d'afficher les erreurs éventuelles d'un programme sur la sortie standard des erreurs. Ce type de sortie n'est défini que dans le monde Unix, et la sortie `err` est identique à la sortie `out` dans le monde Dos.

L'affichage de données

Le principe général, pour l'affichage d'un message, est de placer ce dernier en paramètre de la fonction `System.out.print()`, c'est-à-dire à l'intérieur des parenthèses qui suivent le terme `System.out.print`. Plusieurs possibilités existent quant à la forme et à la syntaxe de ce message, et nous les présentons ci-après.

Affichage de la valeur d'une variable

Soit la variable entière `valeur`. L'affichage de son contenu à l'écran est réalisé par :

```
int valeur = 22 ;
System.out.print(valeur) ;
```

À l'écran, le résultat s'affiche ainsi :

```
22
```

Affichage d'un commentaire

Le fait d'écrire une valeur numérique, sans autre commentaire, n'a que peu d'intérêt. Pour expliquer un résultat, il est possible d'ajouter du texte avant ou après la variable, comme dans l'exemple :

```
System.out.print(" Le montant s'eleve a : " + valeur) ;
```

ou

```
System.out.print(valeur + " correspond au montant total ") ;
```

Pour ajouter un commentaire avant ou après une variable, il suffit de le placer entre guillemets (`" "`) et de l'accrocher à la variable à l'aide du signe +. De cette façon, le compilateur est capable de distinguer le texte à afficher du nom de la variable. Tout caractère placé entre guillemets est un message, alors qu'un mot non entouré de guillemets correspond au nom d'une variable.

En reprenant la même variable `valeur` qu'à l'exemple précédent, le résultat affiché pour le premier exemple est :

```
Le montant s'eleve a : 22
```

Ou encore, pour le deuxième exemple :

```
22 correspond au montant total
```

Affichage de plusieurs variables

On peut afficher le contenu de plusieurs variables en utilisant la même technique. Les commentaires sont placés entre guillemets, et les variables sont précédées, entourées ou suivies du caractère +. Le signe + réunit chaque terme de l'affichage au suivant ou au précédent. Pour afficher le contenu de deux variables :

```
int v = 5, s = 220 ;
```

nous écrivons

```
System.out.print(v + " elements valent au total " + s + " francs ") ;
```

L'exécution de cette instruction a pour résultat :

```
5 elements valent au total 220 francs
```

Affichage de la valeur d'une expression arithmétique

Dans une instruction d'affichage, il est possible d'afficher directement le résultat d'une expression mathématique, sans qu'elle ait été calculée auparavant. Par exemple, nous pouvons écrire :

```
int a = 10, b = 5 ;
System.out.print(a + " fois " + b + " est egal a " + a * b) ;
```

À l'écran, le résultat s'affiche ainsi :

```
10 fois 5 est egal a 50
```

Mais attention ! cette expression est calculée au cours de l'exécution de l'instruction, elle n'est pas mémorisée dans un emplacement mémoire. Le résultat ne peut donc pas être réutilisé dans un autre calcul.

Remarquons, en outre, que l'écriture d'une expression mathématique à l'intérieur de la fonction d'affichage peut être source de confusion pour le compilateur, surtout si

l'expression mathématique comporte un ou plusieurs signes +. En remplaçant, dans l'exemple précédent, le signe * par +, nous obtenons :

```
int a = 10, b = 5 ;
System.out.print(a + " plus " + b + " est egal a " + a + b) ;
```

À l'écran, le résultat s'affiche de la façon suivante :

```
10 plus 5 est egal a 105
```

L'ordinateur ne peut pas afficher la somme de `a` et de `b` parce que, lorsque le signe + est placé dans la fonction d'affichage, il a pour rôle de réunir des valeurs et du texte sur une même ligne d'affichage et non d'additionner deux valeurs. `105` n'est que la réunion de `10` et de `5`. On dit qu'il s'agit d'une opération de **concaténation**.

Pour afficher le résultat d'une addition, il est nécessaire de placer entre parenthèses le calcul à afficher. Par exemple :

```
int a = 10, b = 5 ;
System.out.print(a + " plus " + b + " est egal a " + (a + b)) ;
```

Le résultat à l'écran est :

```
10 plus 5 est egal a 15
```

Affichage d'un texte

Nous pouvons aussi afficher un simple texte sans utiliser de variable :

```
System.out.print("Qui seme le vent recolte la tempete ! ") ;
```

À l'écran, le résultat s'affiche ainsi :

```
Qui seme le vent recolte la tempete !
```

Pour changer de ligne

Remarquons que l'instruction `System.out.print` affiche les informations à la suite de celles qui ont été affichées par un précédent `System.out.print`. Il n'y a pas de passage à la ligne entre deux instructions d'affichage. Ainsi, les instructions :

```
System.out.print("Qui seme le vent ") ;
System.out.print("recolte la tempete ! ") ;
```

ont le même résultat à l'écran que celle de l'exemple précédent :

```
Qui seme le vent recolte la tempete !
```

Pour obtenir un passage à la ligne, il est nécessaire d'utiliser la fonction

```
System.out.println()
```

Ainsi, les instructions

```
System.out.println("Qui seme le vent ") ;
System.out.print("recolte la tempete ! ") ;
```

ont pour résultat :

```
Qui seme le vent
recolte la tempete !
```

Les caractères spéciaux

La table Unicode définit tous les caractères textuels (alphanumériques) et semi-graphiques (*voir, au chapitre 1, « Stocker une information », la section « Les types de base en Java – Catégorie caractère »*). Les caractères spéciaux sont définis entre les 128e et 256e caractères de cette table. Ils correspondent à des caractères n'existant pas sur le clavier mais qui sont néanmoins utiles. Les caractères accentués font aussi partie des caractères spéciaux, les claviers Qwerty américains ne possédant pas ce type de caractères.

Pour afficher un message avec des caractères n'existant pas sur le clavier ou avec des caractères accentués, la méthode consiste à insérer, à l'intérieur du message, le code Unicode du caractère souhaité. Ce code s'obtient en plaçant derrière les caractères `\u00` la valeur hexadécimale de la position du caractère dans la table Unicode. Par exemple, le caractère `A` majuscule est défini en position `65` dans la table Unicode. Son code Unicode s'écrit `\u0041`, car `41` est la valeur hexadécimale de `65`.

L'affichage de caractères accentués et, plus généralement, de tout caractère spécial reste problématique. Surtout si le programme doit fonctionner sur des ordinateurs différents. En effet, les codes de ces caractères font partie des extensions qui diffèrent suivant les pays ou les environnements de travail. Dans ces extensions, les caractères ne sont pas toujours définis à la même position dans la table Unicode. Le caractère `é` est défini en position `234` dans la table Unicode d'Unix, alors qu'il est en position `200` dans la table Unicode du système Mac OS. Les caractères spéciaux, et par conséquent les caractères accentués, n'ont pas toujours un code Unicode identique d'un environnement à un autre.

Par exemple, les caractères `é`, `è` et `ê` ont les codes Unicode suivants :

Environnement	é	è	ê
Unix	`\u00e9`	`\u00e8`	`\u00ea`
Dos	`\u0082`	`\u008a`	`\u0088`
Windows	`\u00e9`	`\u00e8`	`\u00ea`
Mac OS	`\u00c8`	`\u00cb`	`\u00cd`

Le message « `Qui sème le vent récolte la tempête !` » s'écrit donc différemment suivant l'environnement utilisé :

Exemple sous Windows ou Unix

```
System.out.print("Qui s\u00e8me le vent ");
System.out.print("r\u00e9colte la temp\u00eate !");
```

Exemple sous Dos

```
System.out.print("Qui s\u008ame le vent ");
System.out.print("r\u0082colte la temp\u0088te !");
```

Exemple sous MacOS

```
System.out.print("Qui s\u00cbme le vent ");
System.out.print("r\u00c8colte la temp\u00cdte !");
```



Pour connaître le code Unicode d'un caractère donné en fonction de votre environnement de travail, vous pouvez utiliser l'exemple décrit à la section *« La boucle* `for` *»* du chapitre 4, *« Faire des répétitions ».*

La saisie de données

Java est un langage conçu avant tout pour être exécuté dans un environnement Internet et utilisant des programmes essentiellement axés sur le concept d'interface graphique (gestion des boutons, menus, fenêtres, etc.). Dans ce type d'environnement, la saisie de données est gérée par des fenêtres spécialisées, appelées fenêtres de dialogue.

L'objectif de cet ouvrage est d'initier le lecteur au langage Java et, surtout, de lui faire comprendre comment construire et élaborer un programme. Pour cet apprentissage (algorithme et langage), il n'est pas recommandé de se lancer dans l'écriture de programmes utilisant des boutons, des menus et autres fenêtres sans avoir étudié au préalable toute la libraire AWT (*Abstract Windowing Toolkit*) de Java. Cette librairie facilite, il est vrai, la construction d'applications graphiques, mais elle complique et alourdit l'écriture des programmes.

✔ Pour plus de détails sur la librairie AWT, reportez-vous au chapitre 11, *« Dessiner des objets ».*

C'est pourquoi nous avons délibérément choisi de travailler dans un environnement non graphique, plus simple à programmer.

Dans cet environnement, le langage Java propose la fonction `System.in.read()`, qui permet la saisie de données au clavier, sans l'intermédiaire de fenêtres graphiques. Cette fonction est définie dans la bibliothèque `System`, à l'intérieur du sous-ensemble `in`. Elle utilise le programme de lecture au clavier `read()`.

La fonction `System.in.read()` permet de récupérer un et un seul caractère saisi au clavier. Si l'utilisateur souhaite saisir des valeurs ou des noms composés de plusieurs caractères, le programme doit contenir autant d'instructions `System.in.read()` que de caractères à saisir. Le nombre de caractères à saisir variant suivant l'utilisation de l'application, cette fonction n'est pas directement utilisable de cette façon.

La classe `Lire.java`

C'est pourquoi nous proposons au lecteur un ensemble de fonctions de lecture qui permettent de saisir autant de caractères que souhaité. Pour terminer la saisie, il suffit de la valider en appuyant sur la touche `entrée` du clavier. De plus, il existe autant de fonctions de lecture que de types de variables. Il est très facile de saisir des valeurs numériques de type entier (`byte`, `short`, `int` et `long`) ou réel (`float` et `double`) et des caractères de type `char` ou `String`.

Pour ce faire, la technique consiste à utiliser comme nom de fonction le nom `Lire.#()`, où `#` correspond à la première lettre du type de la variable à saisir. Pour saisir un entier,

nous utilisons la fonction `Lire.i()` (`i` étant le premier caractère du mot-clé `int` représentant le type entier). `Lire` est le nom de la bibliothèque des fonctions de saisie de valeurs au clavier. Elle est définie dans le fichier `Lire.java`. Vous trouverez ce fichier dans le CD-Rom livré avec cet ouvrage.

Dans ce fichier, que tout lecteur peut consulter à l'aide d'un éditeur de texte, est défini l'ensemble des fonctions qui facilitent la saisie des données au clavier. Ces fonctions seront étudiées et analysées au fur et à mesure de l'avancement des connaissances, mais pour vous familiariser rapidement avec leur emploi, vous trouverez ci-dessous un programme simple et complet qui utilise toutes les fonctions de saisie proposées par l'auteur.

Exemple : code source complet

```
public class TestLire {
 public static void main (String [] Arg) {
// Déclaration des variables, les noms sont choisis pour une meilleure
// lisibilité du programme, d'autres noms auraient pu être retenus
  byte val_byte ;
  short val_short ;
  int val_int ;
  long val_long ;
  float val_float ;
  double val_double ;
  char val_char ;
  String val_String ;
// Saisir une valeur de type byte
  System.out.println("Entrez un byte : ") ;
  val_byte = Lire.b() ;
// Saisir une valeur de type short
  System.out.println("Entrez un short : ") ;
  val_short = Lire.s() ;
// Saisir une valeur de type int
  System.out.println("Entrez un int : ") ;
  val_int = Lire.i() ;
// Saisir une valeur de type long
  System.out.println("Entrez un long : ") ;
  val_long = Lire.l() ;
// Saisir une valeur de type float
  System.out.println("Entrez un float : ") ;
  val_float = Lire.f() ;
// Saisir une valeur de type double
  System.out.println("Entrez un double : ") ;
  val_double = Lire.d() ;
// Saisir une valeur de type String
  System.out.println("Entrez un String: ") ;
  val_String = Lire.S() ;
// Saisir une valeur de type char
  System.out.println("Entrez un char : ") ;
  val_char = Lire.c() ;
// Afficher les différentes valeurs saisies au clavier
```

```
    System.out.println("vous avez entre le byte : " + val_byte) ;
    System.out.println("vous avez entre le short" + val_short) ;
    System.out.println("vous avez entre l'entier : " + val_int) ;
    System.out.println("vous avez entre le long : " + val_long) ;
    System.out.println("vous avez entre le float : " + val_float) ;
    System.out.println("vous avez entre le double : " + val_double) ;
    System.out.println("vous avez entre le caractere : " + val_char) ;
    System.out.println("vous avez entre le String : " + val_String) ;
  }
}
```

Après la déclaration des variables, le programme demande la saisie de valeurs d'un certain type. L'utilisateur fournit la valeur correspondant au type demandé et valide la saisie en appuyant sur la touche `Entrée` du clavier. Une fois saisies, les valeurs sont affichées à l'écran.

Résultat de l'exécution

Les caractères grisés sont des valeurs choisies par l'utilisateur.

```
Entrez un byte : 100
Entrez un short : -30560
Entrez un int : 125698
Entrez un long : 98768765
Entrez un float : 3.14159
Entrez un double : 123.876453097432
Entrez un String: Exemple
Entrez un char : A
vous avez entre le byte :100
vous avez entre le short :-30560
vous avez entre l'entier : 125698
vous avez entre le long : 98768765
vous avez entre le float : 3.14159
vous avez entre le double : 123.876453097432
vous avez entre le caractere : A
vous avez entre le String : Exemple
```

Résumé

Pour communiquer une information, l'ordinateur affiche un message à l'écran. On dit qu'il réalise une opération de **sortie** (out) ou d'écriture de données. À l'inverse, lorsque l'utilisateur communique des données au programme par l'intermédiaire du clavier, il effectue une opération d'**entrée** (in) ou de lecture de données.

Dans le langage Java, les opérations de sortie sont réalisées grâce à l'instruction `System.out.print()`, qui permet d'afficher des informations à l'écran.

Par exemple, l'instruction :

```
System.out.print(F + " francs valent " + E + " euros") ;
```

affiche à l'écran le contenu de la variable `F`, suivi du texte « `francs valent` », puis le contenu de la variable `E`, suivi du texte « `euros` ».

Pour distinguer le commentaire du nom de variable, le commentaire est placé entre guillemets. Le contenu de la variable est affiché en réunissant la variable au commentaire à l'aide du signe +.

Pour afficher des résultats sur plusieurs lignes, il convient d'utiliser l'instruction :

```
System.out.println()
```

Dans le langage Java, les opérations d'entrée ne sont pas aussi simples d'emploi du fait qu'elles sont le plus souvent réalisées à l'aide de fenêtres graphiques générant des programmes plus complexes à écrire.

C'est la raison pour laquelle l'auteur propose un ensemble de fonctions de lecture qui permettent la saisie de valeurs de tout type. Par exemple, pour saisir un entier, il suffit d'utiliser la fonction `Lire.i()` (`i` étant le premier caractère du mot-clé `int` représentant le type entier). Les fonctions de lecture ont pour nom d'appel :

`Lire.b()` ; pour saisir une valeur de type `byte` ;

`Lire.s()` ; pour saisir une valeur de type `short` ;

`Lire.i()` ; pour saisir une valeur de type `int` ;

`Lire.l()` ; pour saisir une valeur de type `long` ;

`Lire.f()` ; pour saisir une valeur de type `float` ;

`Lire.d()` ; pour saisir une valeur de type `double` ;

`Lire.S()` ; pour saisir une valeur de type `String` ;

`Lire.c()` ; pour saisir une valeur de type `char`.

Exercices

Comprendre les opérations de sortie

2.1 Soit un programme Java contenant les déclarations :

```
int i = 223, j = 135 ;
float a = 335.5f, b = 20.5f ;
char R = 'R', T = 'T' ;
```

Décrivez l'affichage généré par chacune des instructions suivantes :

```
System.out.println("Vous avez entre : " + i) ;
System.out.println("Pour un montant de "+ a + " le total vaut : "+ i + j) ;
System.out.print("Apres reduction de " + b + " %, vous gagnez : ") ;
System.out.println( (a*b)/100 + " euros") ;
System.out.print(" La variable R = " + R + " et T = " + T) ;
```

2.2 En tenant compte des déclarations de variables suivantes, écrivez les instructions `System.out.print()` de façon à obtenir l'affichage suivant :

double x = 4, y = 2 ;	double x = 9, y = 3 ;
x = 4.0 et y = 2.0 Racine carrée de 4.0 = 2.0 4.0 a la puissance 2.0 = 16.0	x = 9.0 et y = 3.0 Racine carrée de 9.0 = 3.0 9.0 a la puissance 3.0 = 729.0

✔ Notez que la racine carrée de `x` s'obtient par la fonction `Math.sqrt(x)` et que a^b se calcule avec la méthode `Math.pow(a,b)`.

Comprendre les opérations d'entrée

2.3 Pour chacun des deux programmes suivants, et compte tenu des informations fournies par l'utilisateur, quelles sont les valeurs affichées à l'écran ?

L'utilisateur fournit au clavier 2, puis 3, puis 4	L'utilisateur fournit au clavier 2

`int X,  Y ;`
`X = Lire.i() ;`
`Y = Lire.i() ;`
`X = Lire.i() ;`
`X = X+Y ;`
`System.out.print(" X = " + X) ;`
`System.out.print(" Y = " + Y) ;`
	`int X,  Y ;`
	`X = Lire.i() ;`
	`Y = 0 ;`
	`X = X+Y ;`
	`System.out.println(" X = " + X) ;`
	`System.out.println(" Y = " + Y) ;`

Observer et comprendre la structure d'un programme Java

2.4 En prenant exemple sur la structure suivante, écrivez un programme `Euro` qui convertisse des francs en euros. (Rappel : 1 euro = 6,559 57 francs) :

```
public class .............................// Donner un nom à la classe
{
  public static void main(String [] argument)
  {
   // Déclarer les variables représentant les francs et les euros
   // ainsi que le taux de conversion
   ......................................................
   // Afficher et Saisir le nombre de francs
   ......................................................
   ......................................................
   // Calculer le nombre d'euros
   ......................................................
   // Afficher le résultat suivant l'exemple donné ci-dessous
   ......................................................
  }
}
```

L’affichage du résultat se fera sous la forme suivante :

```
Nombre de francs      :      120
Conversion F/E        :      6,559 57
Nombre d’euros        :      18,293
```

Le projet « Gestion d’un compte bancaire »

Afficher le menu principal ainsi que ses options

L’objectif du premier programme est d’écrire toutes les instructions qui permettent l’affichage des menus définis dans le cahier des charges décrit au chapitre introductif, « Naissance d’un programme », ainsi que la saisie des données demandées. Le programme construit affiche tous les messages de toutes les options, sans contrôle sur le choix de l’utilisateur.

- Le menu principal s’affiche de la façon suivante :

```
1. Creer un compte
2. Afficher un compte
3. Creer une ligne comptable
4. Sortir
5. De l’aide
Votre choix :
```

- Une fois le menu affiché, le programme attend la saisie du choix de l’utilisateur.
- L’option `1` du menu principal a pour affichage :

```
Type du compte [Types possibles : courant, joint, épargne] :
Numero du compte :
Première valeur creditee :
Taux de placement :
```

- L’option `2` réalise les opérations suivantes :
  - Affiche la demande de saisie du numéro du compte que l’utilisateur souhaite consulter.
  - Saisit le numéro de compte.
- L’option `3` affiche : « `option non programmee` ».
- L’option `4` termine l’exécution du programme. Pour cela, utilisez la fonction Java `System.exit(0) ;`.
- Avec l’option `5`, le programme affiche une ligne d’explication pour chaque option du menu principal.

3

Faire des choix

Une fois les variables définies et les valeurs stockées en mémoire, l'ordinateur est capable de les **tester** ou de les **comparer** de façon à réaliser une instruction plutôt qu'une autre, suivant le résultat de la comparaison.

Le programme n'est alors plus exécuté de façon séquentielle (de la première ligne jusqu'à la dernière). L'ordre est rompu, une ou plusieurs instructions étant ignorées en fonction du résultat du test. Le programme peut s'exécuter, en tenant compte de contraintes imposées par le programmeur.

Dans ce chapitre, nous abordons la notion de choix ou de test, en reprenant l'algorithme du café chaud, pour le transformer en un algorithme du café chaud sucré **ou** non (*« L'algorithme du café chaud, sucré ou non »*).

Ensuite, à la section « L'instruction `if-else` », nous étudions la structure `if-else` proposée par le langage Java, qui permet de réaliser des choix.

Enfin, à la section « L'instruction `switch`, ou comment faire des choix multiples », nous examinons le concept de choix multiples par l'intermédiaire de la structure `switch`.

L'algorithme du café chaud, sucré ou non

Pour mieux comprendre la notion de choix, nous allons reprendre l'algorithme du café chaud pour le transformer en `algorithme du café chaud, sucré ou non`. L'énoncé ainsi transformé nous oblige à modifier la liste des objets manipulés, ainsi que celle des opérations à réaliser.

Définition des objets manipulés

Pour obtenir du café sucré, nous devons ajouter à notre liste un nouvel ingrédient, le sucre, et un nouvel ustensile, la petite cuillère.

```
café moulu
filtre
eau
cafetière électrique
tasse
électricité
table
sucre
petite cuillère
```

Liste des opérations

De la même façon, nous devons modifier la liste des opérations, de façon qu'elle prenne en compte les nouvelles données :

```
Verser l'eau dans la cafetière, le café dans la tasse, le café dans le filtre.
Prendre du café moulu, une tasse, de l'eau, une cafetière électrique,
➥un filtre, un morceau de sucre, une petite cuillère.
Brancher, allumer ou éteindre la cafetière électrique.
Attendre que le café soit prêt.
Poser la tasse, la cafetière sur la table, le filtre dans la cafetière, le sucre
➥dans la tasse, la petite cuillère dans la tasse.
```

Ordonner la liste des opérations

Ainsi modifiée, la liste des opérations doit être réordonnée afin de rechercher le moment le mieux adapté pour ajouter les nouvelles opérations :

- En décidant de prendre le sucre et la petite cuillère en même temps que le café et le filtre, nous plaçons les nouvelles instructions « `prendre…` » entre les instructions 2 et 3 définies à la section « Ordonner la liste des opérations » du chapitre introductif, « Naissance d'un programme ».
- En décidant de poser le sucre et la petite cuillère dans la tasse avant d'y verser le café, nous écrivons les nouvelles instructions « `poser…` » avant l'instruction 15 du même exemple.

Nous obtenons la liste des opérations suivantes :

```
0.   Prendre une cafetière.
1.   Poser la cafetière sur la table.
2.   Prendre du café.
3.   Prendre un morceau de sucre.
4.   Prendre une petite cuillère
5.   Prendre un filtre.
6.   Verser le café dans le filtre.
```

```
7.   Prendre de l'eau.
8.   Verser l'eau dans la cafetière.
9.   Brancher la cafetière.
10.  Allumer la cafetière.
11.  Attendre que le café soit prêt.
12.  Prendre une tasse.
13.  Poser la tasse sur la table.
14.  Poser le sucre dans la tasse.
15.  Poser la petite cuillère dans la tasse.
16.  Eteindre la cafetière.
17.  Verser le café dans la tasse.
```

Écrite ainsi, cette marche à suivre nous permet d'obtenir un café chaud sucré. Elle ne nous autorise pas à choisir entre sucré ou non. Pour cela, nous devons introduire un test, en posant une condition devant chaque instruction concernant la prise du sucre, c'est-à-dire :

```
0.   Prendre une cafetière.
1.   Poser la cafetière sur la table.
2.   Prendre du café.
3.   Si (café sucré) Prendre un morceau de sucre.
4.   Si (café sucré) Prendre une petite cuillère.
5.   Prendre un filtre.
6.   Verser le café dans le filtre.
7.   Prendre de l'eau.
8.   Verser l'eau dans la cafetière.
9.   Brancher la cafetière.
10.  Allumer la cafetière.
11.  Attendre que le café soit prêt.
12.  Prendre une tasse.
13.  Poser la tasse sur la table.
14.  Si (café sucré) Poser le sucre dans la tasse.
15.  Si (café sucré) Poser la petite cuillère dans la tasse.
16.  Eteindre la cafetière.
17.  Verser le café dans la tasse.
```

Dans cette situation, nous obtenons du café sucré ou non, selon notre choix. Remarquons cependant que le test `Si (café sucré)` est identique pour les instructions `3`, `4`, `14` et `15`. Pour cette raison, et sachant que chaque test représente un coût en termes de temps d'exécution, il est conseillé de regrouper au même endroit toutes les instructions relatives à un même test.

C'est pourquoi nous distinguons deux blocs d'instructions distincts :

- les instructions soumises à la condition de café sucré (`II Préparer le sucre`) ;
- les instructions réalisables quelle que soit la condition (`I Préparer le café`).

Dans ce cas, la nouvelle solution s'écrit :

Instructions	Bloc d'instructions

`0. Prendre une cafetière.`
`1. Poser la cafetière sur la table.`
`2. Prendre du café.`
`3. Prendre un filtre.`
`4. Verser le café dans le filtre.`
`5. Prendre de l'eau.`
`6. Verser l'eau dans la cafetière.`
`7. Brancher la cafetière.`
`8. Allumer la cafetière.`
`9. Attendre que le café soit prêt.`
`10. Prendre une tasse.`
`11. Poser la tasse sur la table.`
`12. Eteindre la cafetière.`
`13. Verser le café dans la tasse.`
	I Préparer le café

`Si (café sucré)`	

`1. Prendre un morceau de sucre.`
`2. Prendre une petite cuillère.`
`3. Poser le sucre dans la tasse.`
`4. Poser la petite cuillère dans la tasse.`
	II Préparer le sucre

La réalisation du bloc `I Préparer le café` nous permet d'obtenir du café chaud. Ensuite, en exécutant le test `Si (café sucré)`, deux solutions sont possibles :

- La proposition `(café sucré)` est vraie, et alors les instructions 1 à 4 du bloc `II Préparer le sucre` sont exécutées. Nous obtenons du café chaud sucré.
- La proposition `(café sucré)` est fausse, et les instructions qui suivent ne sont pas exécutées. Nous obtenons un café non sucré.

Pour programmer un choix, nous avons écrit une condition devant les instructions concernées. En programmation, il en est de même. Le langage Java propose plusieurs instructions de test, à savoir la structure `if-else`, que nous étudions ci-après, et la structure `switch` que nous analysons à la section « L'instruction `switch`, ou comment faire des choix multiples », un peu plus loin dans ce chapitre.

L'instruction `if-else`

L'instruction `if-else` se traduit en français par les termes `si-sinon`. Elle permet de programmer un choix, en plaçant derrière le terme `if` une condition, comme nous avons placé une condition derrière le terme `si` de l'algorithme du café chaud, sucré ou non.

L'instruction `if-else` se construit de la façon suivante :

- en suivant une syntaxe, ou forme, précise du langage Java (*voir « Syntaxe d'*if-else ») ;
- en précisant la condition à tester (*voir « Comment écrire une condition »*).

Nous présentons en fin de cette section un exemple de programme qui recherche la plus grande des deux valeurs saisies au clavier (*voir « Rechercher le plus grand de deux éléments »*).

Syntaxe d'`if-else`

L'écriture de l'instruction `if-else` obéit aux règles de syntaxe suivantes :

```
if     (condition)     // si la condition est vraie
{                      // faire
    plusieurs instructions ;
}                      // fait
else                   // sinon (la condition ci-dessus est fausse)
{                      //faire
    plusieurs instructions ;
}                      //fait
```

- Si la condition située après le mot-clé `if` et placée obligatoirement entre parenthèses est vraie, alors les instructions placées dans le bloc défini par les accolades ouvrante et fermante immédiatement après sont exécutées.
- Si la condition est fausse, alors les instructions définies dans le bloc situé après le mot-clé `else` sont exécutées.

De cette façon, un seul des deux blocs peut être exécuté à la fois, selon que la condition est vérifiée ou non.

Remarquons que :

- La ligne d'instruction `if(`condition`)`ou `else` ne se termine jamais par un point-virgule (`;`).
- Les accolades `{` et `}` définissent un bloc d'instructions. Cela permet de regrouper ensemble toutes les instructions relatives à un même test.
- L'écriture du bloc `else` n'est pas obligatoire. Il est possible de n'écrire qu'un bloc `if` sans programmer d'instruction dans le cas où la condition n'est pas vérifiée (comme dans l'algorithme du café chaud, sucré ou non). En d'autres termes, il peut y avoir des `if` sans `else`.
- S'il existe un bloc `else`, celui-ci est obligatoirement « accroché » à un `if`. Autrement dit, il ne peut y avoir d'`else` sans `if`.
- Le langage Java propose une syntaxe simplifiée lorsqu'il n'y a qu'une seule instruction à exécuter dans l'un des deux blocs `if` ou `else`. Dans ce cas, les accolades ouvrante et fermante ne sont pas obligatoires :

```
if     (condition)     une seule instruction ;
else         une seule instruction ;
```

ou :

```
if     (condition)
{                              // faire
    plusieurs instructions ;
}                              // fait
else         une seule instruction ;
```

ou encore :

```
if     (condition)     une seule instruction ;
else
{                              // faire
```

```
    plusieurs instructions ;
}                            // fait
```

Une fois connue la syntaxe générale de la structure `if-else`, nous devons écrire la condition (placée entre parenthèses, juste après `if`) permettant à l'ordinateur d'exécuter le test.

Comment écrire une condition

L'écriture d'une condition en Java fait appel aux notions d'opérateurs relationnels et conditionnels.

Les opérateurs relationnels

Une condition est formée par l'écriture de la comparaison de deux expressions, une expression pouvant être une valeur numérique ou une expression arithmétique. Pour comparer deux expressions, le langage Java dispose de six symboles représentant les opérateurs relationnels traditionnels en mathématiques.

Opérateur	Signification pour des valeurs numériques	Signification pour des valeurs de type caractère
= =	égal	identique
<	inférieur strictement	plus petit dans l'ordre alphabétique
<=	inférieur ou égal	plus petit ou identique dans l'ordre alphabétique
>	supérieur strictement	plus grand dans l'ordre alphabétique
>=	supérieur ou égal	plus grand ou identique dans l'ordre alphabétique
!=	différent	différent

Un opérateur relationnel permet de comparer deux expressions de même type. Il n'est pas possible de comparer un réel avec un entier ou un entier avec un caractère.

Lorsqu'il s'agit de comparer deux expressions composées d'opérateurs arithmétiques (`+ - * / %`), les opérateurs relationnels sont moins prioritaires par rapport aux opérateurs arithmétiques. De cette façon, les expressions mathématiques sont d'abord calculées avant d'être comparées.

Notons que pour tester l'égalité entre deux expressions, nous devons utiliser le symbole == et non pas un simple =. En effet, en Java, le signe = n'est pas un signe d'égalité au sens de la comparaison mais le signe de l'affectation, qui permet de placer une valeur dans une variable.

Exemple

```
int a = 3, b = 5 ;
char lettre = 'i', car = 'j' ;
```

- La condition `(a != b)` est vraie car 3 est différent de 5.
- La condition `(a + 2 == b)` est vraie car 3 + 2 vaut 5.
- La condition `(a + 8 < 2 * b)`est fausse car 3 + 8 est plus grand que 2 * 5.

- La condition `(lettre <= car)` est vraie car le caractère `'i'` est placé avant `'j'` dans l'ordre alphabétique.
- La condition `(lettre == 'w')` est fausse car le caractère `'i'` est différent du caractère `'w'`.

Les opérateurs logiques

Les opérateurs logiques sont utilisés pour associer plusieurs conditions simples et, de cette façon, créer des conditions multiples en un seul test. Il existe trois grands opérateurs logiques, symbolisés par les caractères suivants :

Opérateur	Signification
`!`	NON logique
`&&`	ET logique
`||`	OU logique

Exemples

```
int x = 3, y = 5, z = 2, r = 6 ;
```

- Sachant que la condition `(x < y) && (z < r)` est vraie si les deux expressions `(x < y)` **et** `(z < r)` sont toutes les deux vraies et devient fausse si l'une des deux expressions est fausse, l'expression donnée en exemple est vraie. En effet `(3 < 5)` est vraie et `(2 < 6)` est vraie.
- Sachant que la condition `(x > y) || (z < r)` est vraie si l'une des expressions `(x > y)` **ou** `(z < r)` est vraie et devient fausse si les deux expressions sont fausses, l'expression donnée en exemple est vraie car `(3 > 5)` est fausse, mais `(2 < 6)` est vraie.
- Sachant que la condition `!(z < r)` est vraie si l'expression `(z < r)` est fausse et devient fausse si l'expression est vraie, alors l'expression donnée en exemple est fausse car `(2 < 6)` est vraie.

Rechercher le plus grand de deux éléments

Pour mettre en pratique les notions théoriques abordées aux deux sections précédentes, nous allons écrire un programme qui affiche, dans l'ordre croissant, deux valeurs entières saisies au clavier et recherche la plus grande des deux. Pour cela, nous devons :

1. Demander la saisie de deux valeurs au clavier.
2. Tester si la première valeur saisie est plus grande que la seconde.
   a. Si tel est le cas :
      – afficher dans l'ordre croissant, en affichant la seconde valeur saisie puis la première ;
      – stocker la plus grande des valeurs dans une variable spécifique, soit la première valeur.

b. Sinon :

– afficher dans l'ordre croissant, en affichant la première valeur saisie puis la seconde ;
– stocker la plus grande des valeurs dans une variable spécifique, soit la seconde valeur.

3. Afficher la plus grande des valeurs.

Nous devons, dans un premier temps, déclarer trois variables entières, deux pour les valeurs à saisir et une pour stocker la plus grande des deux. Nous écrivons l'instruction de déclaration suivante :

```
int première, deuxième, laPlusGrande ;
```

1. La saisie des deux valeurs est ensuite réalisée par (*voir le chapitre 2, « Communiquer une information »*) :

```
System.out.print("Entrer une valeur :") ;
première =  Lire.i() ;
System.out.print("Entrer une deuxieme valeur :") ;
deuxième =  Lire.i() ;
```

2. Pour tester si la première valeur saisie est plus grande que la seconde, l'instruction `if` s'écrit :

```
if (première > deuxième)
```

a. Deux instructions composent ce test : l'affichage dans l'ordre croissant puis le stockage de la plus grande valeur. Il est donc nécessaire de les placer dans un bloc défini par une `{` ouvrante et une `}` fermante :

```
{
   // Afficher les valeurs dans l'ordre croissant
   System.out.println(deuxième + "  " + première) ;
   // Stocker la plus grande dans une variable spécifique
   laPlusGrande = première ;
}
```

b. De la même façon, le cas contraire est décrit par l'instruction `else` et est composé de deux instructions. Nous avons donc :

```
else
{
  // Afficher les valeurs dans l'ordre croissant
  System.out.println(première + " " + deuxième) ;
  // Stocker la plus grande dans une variable spécifique
  laPlusGrande = deuxième ;
}
```

3. Nous affichons enfin la plus grande valeur par l'instruction :

```
System.out.println("La plus grande valeur est : " + laPlusGrande) ;
```

Ce message est affiché dans tous les cas, et l'instruction est donc placée en dehors de toute structure conditionnelle.

Pour finir, le programme est placé dans une fonction `main()` et une classe, que nous appelons `Maximum`, puisqu'il s'agit ici de trouver la valeur maximale de deux valeurs. De cette façon, le programme peut être compilé et exécuté.

Exemple : code source complet

```
public class Maximum  // Le fichier s'appelle Maximum.java
{
  public static void main (String [] parametre)
  {
   int première, deuxième, laPlusGrande ;
   System.out.println("Entrer une valeur :") ;
   première =  Lire.i() ;
   System.out.println("Entrer une deuxieme valeur :") ;
   deuxième =  Lire.i() ;
   if (première > deuxième)
   {
     System.out.println(deuxième + "  " + première) ;
     laPlusGrande = première ;
   }
   else
   {
     System.out.println(première + " " + deuxième) ;
     laPlusGrande = deuxième ;
   }
  System.out.println("La plus grande valeur est : " + laPlusGrande) ;
  } // Fin du main ()
} // Fin de la Class Maximum
```

Résultat de l'exécution

(Les caractères grisés sont des valeurs choisies par l'utilisateur.)

```
Entrer une valeur : 3
Entrer une deuxieme valeur : 5
3  5
La plus grande valeur est : 5
```

La première valeur étant plus petite que la seconde, le programme exécute les instructions placées dans le bloc `else`.

Deux erreurs à éviter

Deux types d'erreurs sont à éviter par le programmeur débutant. Il s'agit des erreurs issues d'une mauvaise construction des blocs `if` ou `else` et d'un placement incorrect du point-virgule.

La construction de blocs

Reprenons l'exemple précédent en l'écrivant comme suit :

```
if (première > deuxième)
     System.out.println(deuxième + "  " + première) ;
     laPlusGrande = première ;
```

```
else
   {
     System.out.println(première+" "+deuxième) ;
     laPlusGrande = deuxième ;
   }
```

En exécutant pas à pas cet extrait de programme, nous remarquons qu'il n'y a pas d'accolade ({) ouvrante derrière l'instruction `if`. Cette dernière ne possède donc pas de bloc composé de plusieurs instructions. Seule l'instruction d'affichage `System.out.println(deuxième + "  " + première) ;` se situe dans `if`. L'exécution d'`if` s'achève donc juste après l'affichage des valeurs dans l'ordre croissant.

Ensuite, l'instruction `lePlusGrand = première ;` est théoriquement exécutée en dehors de toute condition. Cependant, l'instruction suivante est `else,` alors que l'instruction `if` s'est achevée précédemment. Le compilateur ne peut attribuer ce `else` à un `if`. Il y a donc erreur de compilation du type `'else' without 'if'`.

De la même façon, il y a erreur de compilation lorsque le programme est construit sur la forme suivante :

```
if (première > deuxième)
{....
}
Leplusgrand = première ;
else
{...
}
```

Le point-virgule

Dans le langage Java, le point-virgule constitue une instruction à part entière, qui représente l'instruction vide. Par conséquent, écrire le programme suivant ne provoque aucune erreur à la compilation :

```
if (première > deuxième) ;
     System.out.println(deuxième + "  " + première) ;
```

L'exécution de cet extrait de programme a pour résultat :

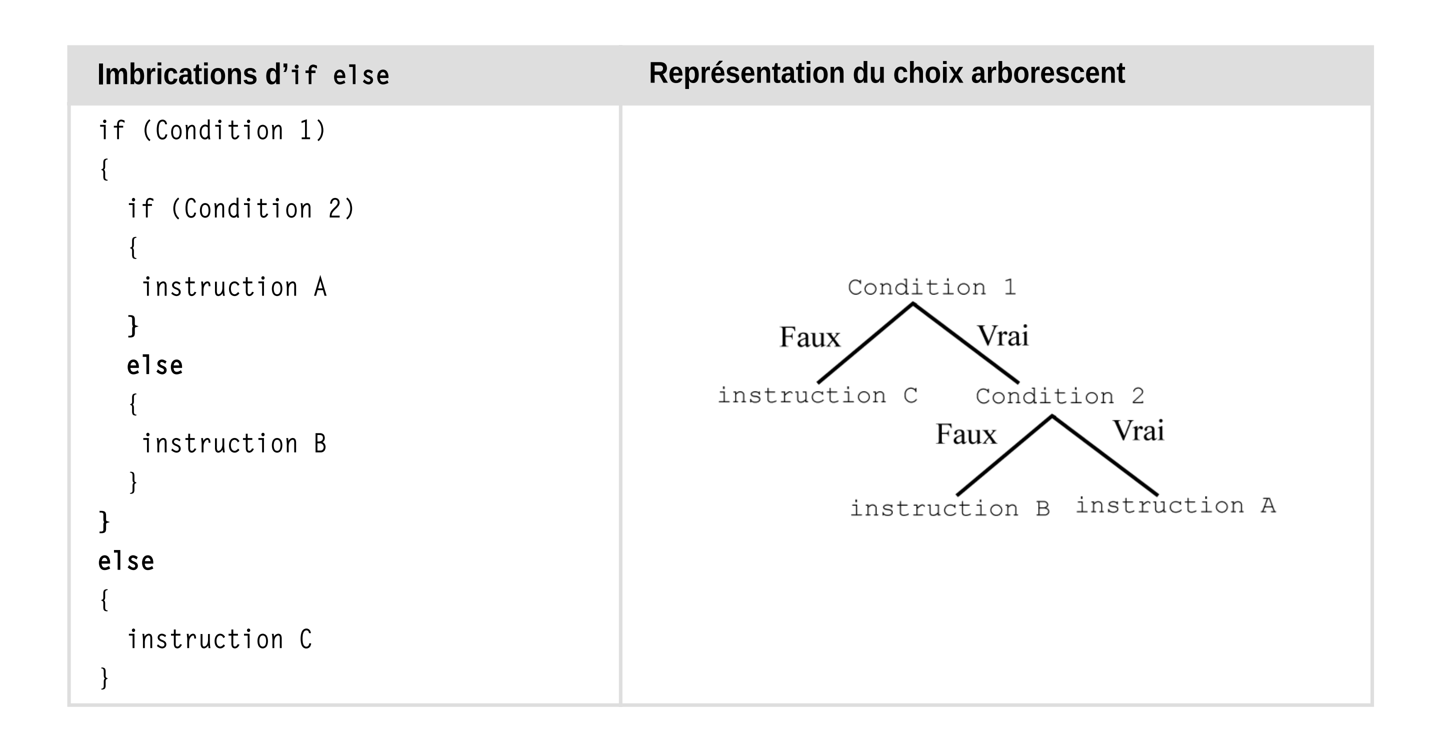
Si `première` est plus grand que `deuxième`, l'ordinateur exécute le `;` (point-virgule) situé immédiatement après la condition, c'est-à-dire rien. L'instruction `if` est terminée, puisqu'il n'y a pas d'accolades ouvrante et fermante. Seule l'instruction `;` est soumise à `if`.

Le message affichant les valeurs par ordre croissant ne fait pas partie du test. Il est donc affiché, quelles que soient les valeurs de `première` et `deuxième`.

Des `if-else` *imbriqués*

Dans le cas de choix arborescents – un choix étant fait, d'autres choix sont à faire, et ainsi de suite –, il est possible de placer des structures `if-else` à l'intérieur d'`if-else`. On dit alors que les structures `if-else` sont imbriquées les unes dans les autres.

Lorsque ces imbrications sont nombreuses, il est possible de les représenter à l'aide d'un graphique de structure arborescente, dont voici un exemple :



Imbrications d'`if else`	Représentation du choix arborescent

```
if (Condition 1)
{
  if (Condition 2)
  {
   instruction A
  }
  else
  {
   instruction B
  }
}
else
{
  instruction C
}
```


Quand il y a moins d'`else` que d'`if`

Une instruction `if` peut ne pas contenir d'instruction `else`. Dans de tels cas, il peut paraître difficile de savoir à quel `if` est associé le dernier `else`. Comparons les deux exemples suivants :

Imbrications d'`if else`	Arbre des choix

```
if (Condition 1)
{
  if (Condition 2)
  {
    if (Condition 3)
    {
      instruction A
    }
    else
    {
      instruction B
    }
  }
}
else
{
  instruction C
}
```

Condition 1
Faux
Vrai
instruction C
Condition 2
Vrai
Condition 3
Faux
Vrai
instruction B
instruction A

Imbrications d'`if else`	Arbre des choix

`if (Condition 1)`
`{`
`  if (Condition 2)`
`  {`
`    if (Condition 3)`
`    {`
`      instruction A`
`    }`
`    else`
`    {`
`      instruction B`
`    }`
`  }`
`  else`
`  {`
`    instruction C`
`  }`
`}`
	Condition 1
	Vrai
	Condition 2
	Faux Vrai
	instruction C Condition 3
	Faux Vrai
	instruction B instruction A

Du premier au deuxième exemple, par le jeu des fermetures d'accolades, le dernier bloc `else` est déplacé d'un bloc vers le haut. Ce déplacement modifie la structure arborescente. Les algorithmes associés ont des résultats totalement différents.

Pour déterminer une relation `if-else`, remarquons qu'un « bloc `else` » se rapporte toujours au dernier « bloc `if` » rencontré, auquel un `else` n'a pas encore été attribué.

Les blocs `if` et `else` étant délimités par les accolades ouvrantes et fermantes, il est conseillé, pour éviter toute erreur, de bien relier chaque parenthèse ouvrante avec sa fermante.

L'instruction `switch`, ou comment faire des choix multiples

Lorsque le nombre de choix possible est plus grand que deux, l'utilisation de la structure `if-else` devient rapidement fastidieuse. Les imbrications des blocs demandent à être vérifiées avec précision, sous peine d'erreur de compilation ou d'exécution.

C'est pourquoi, le langage Java propose l'instruction `switch` (traduire par selon, ou suivant), qui permet de programmer des choix multiples selon une syntaxe plus claire.

Construction du `switch`

L'écriture de l'instruction `switch` obéit aux règles de syntaxe suivantes :

```
switch (valeur)
{
  case étiquette 1 :
           // Une ou plusieurs instructions
```

```
    break ;
    case étiquette 2 :
    case étiquette 3 :
              // Une ou plusieurs instructions
    break ;
    default :
              // Une ou plusieurs instructions
}
```

La variable `valeur` est évaluée. Suivant cette valeur, le programme recherche l'`étiquette` correspondant à la valeur obtenue et définie à partir des instructions `case étiquette` .

- Si le programme trouve une étiquette correspondant au contenu de la variable `valeur`, il exécute la ou les instructions qui suivent l'étiquette, jusqu'à rencontrer le mot-clé `break`.
- S'il n'existe pas d'étiquette correspondant à `valeur`, alors le programme exécute les instructions de l'étiquette `default`

D'une manière générale, remarquons que :

- Le type de la variable `valeur` ne peut être que `char` ou `int`, `byte`, `short` ou `long`. Il n'est donc pas possible de tester des valeurs réelles ou des mots.
- Une étiquette peut contenir aucune, une ou plusieurs instructions.
- L'instruction `break` permet de sortir du bloc `switch`. S'il n'y a pas de `break` pour une étiquette donnée, le programme exécute les instructions de l'étiquette suivante.

Calculer le nombre de jours d'un mois donné

Pour mettre en pratique les notions théoriques abordées à la section précédente, nous allons écrire un programme qui calcule et affiche le nombre de jours d'un mois donné.

Le nombre de jours dans un mois peut varier entre les valeurs `28`, `29`, `30` ou `31`, suivant le mois et l'année. Les mois de janvier, mars, mai, juillet, août, octobre et décembre sont des mois de `31` jours. Les mois d'avril, juin, septembre et novembre sont des mois de `30` jours. Seul le mois de février est particulier, puisque son nombre de jours est de `29` jours pour les années bissextiles et de `28` jours dans le cas contraire. Sachant cela, nous devons :

- Demander la saisie au clavier du numéro du mois ainsi que de l'année recherchée.
- Créer autant d'étiquettes qu'il y a de mois dans une année, c'est-à-dire `12`. Compte tenu du fonctionnement de la structure `switch`, chaque étiquette est une valeur entière correspondant au numéro du mois de l'année (`1` pour janvier, `2` pour février, etc.).
- Regrouper les étiquettes relatives aux mois à `31` jours et stocker cette dernière valeur dans une variable spécifique.
- Regrouper les étiquettes relatives aux mois à `30` jours et stocker cette dernière valeur dans une variable spécifique.
- Pour l'étiquette relative au mois de février, tester la valeur de l'année pour savoir si l'année concernée est bissextile ou non. Une année est bissextile tous les `quatre` ans, sauf lorsque le millésime est divisible par `100` et non pas par `400`. En d'autres termes, pour qu'une année soit bissextile, il suffit que l'année soit un nombre divisible par 4 et

non divisible par `100` ou alors par `400`. Dans tous les autres cas, l'année n'est pas bissextile.

Compte tenu de toutes ces remarques, nous devons dans un premier temps déclarer trois variables entières, une pour représenter le mois, la deuxième l'année, et la troisième le nombre de jours par mois. Sachant que le mois et le nombre de jours par mois ne dépassent jamais la valeur `127`, nous pouvons les déclarer de type `byte`. Pour l'année, le type `short` suffit (à moins d'être très optimiste et de vouloir éviter le bug de l'an 32767 !), puisque les valeurs de ce type peuvent aller jusqu'à `32767`.

Exemple : code source complet

```
public class JourParMois  // Le fichier s'appelle JourParMois.java
{
  public static void main (String [] parametre)
  {
   byte mois, nbjours = 0 ;
   short année ;
   System.out.println("De quel mois s'agit-il ? :") ;
   mois =  Lire.b() ;
   System.out.println("De quelle annee ? :") ;
   année =  Lire.s() ;
   switch(mois)
   {
     case 1 : case 3 :         // Pour les mois à 31 jours
     case 5 : case 7 :
     case 8 : case 10 :
     case 12 :
            nbjours = 31 ;
            break ;
     case 4 : case 6 :         // Pour les mois à 30 jours
     case 9 : case 11 :
            nbjours  = 30 ;
            break ;
     case 2 :                   // Pour le cas particulier du mois de février
          if (année % 4 == 0 && année % 100 != 0 || année % 400 == 0)
               nbjours  = 29 ;
          else  nbjours  = 28 ;
          break ;
     default :                 // En cas d'erreur de frappe
              System.out.println("Impossible, ce mois n'existe pas ") ;
              System.exit(0) ;
   }
   System.out.print(" En " + annee + ", le mois n° " + mois) ;
   System.out.println(" a " + nbjours + " jours ") ;
  } // Fin du main()
} // Fin de la class JourParMois
```

Résultat de l'exécution

Les caractères grisés sont des valeurs choisies par l'utilisateur.

Exécution 1

```
De quel mois s'agit-il ? : 5
De quelle annee ? : 1999
En 1999 le mois n° 5 a 31 jours
```

Le programme recherche l'étiquette `5`. Il exécute les instructions qui suivent jusqu'à rencontrer un `break`. Pour l'étiquette `5`, le programme exécute les instructions des étiquettes `7`, `8`, `10` et `12` car ces étiquettes ne possèdent ni instructions, ni `break`. Seule l'étiquette `12` possède une instruction, qui affecte la valeur `31` à la variable `nbjours`. L'instruction `break` qui suit permet de sortir de la structure `switch`. Le programme exécute enfin l'instruction située immédiatement après le `switch`, c'est-à-dire l'affichage du message annonçant le résultat.

Exécution 2

```
De quel mois s'agit-il ? : 2
De quelle annee ? : 2000
En 2000 le mois n° 2 a 29 jours
```

Ici, le programme va directement à l'étiquette `2`, qui est composée d'un test sur l'année pour savoir si l'année est bissextile. Une année est bissextile lorsque son millésime est divisible par `4`, à l'exception des années dont le millésime est divisible par `100` et non pas par `400`. La valeur `2000` est divisible par `4`, `100` et `400` puisque le reste de la division entière (`%`) de 2000 par `4`, `100` ou `400` est nul. La variable `nbjours` prend donc la valeur `29`. Le programme sort ensuite du `switch` grâce à l'instruction `break` qui suit et exécute pour finir l'affichage du résultat.

Exécution 3

```
De quel mois s'agit-il ? : 15
De quelle annee ? : 1999
Impossible, ce mois n'existe pas
```

L'étiquette `15` n'étant pas définie dans le bloc `switch`, le programme exécute les instructions qui composent l'étiquette `default`. Le programme affiche un message d'erreur et termine son exécution grâce à l'instruction `System.exit(0) ;`

Remarquons que grâce à l'étiquette `default`, le programme connaît les instructions à exécuter dans le cas de choix « anormaux » (erreur de frappe, par exemple, ou valeur saisie n'entrant pas dans l'intervalle des valeurs possibles traitées par le programme). De cette façon, il devient possible de prévenir d'éventuelles erreurs pouvant causer l'arrêt brutal de l'exécution du programme.

Comment choisir entre `if-else` *et* `switch` ?

La structure `switch` ne permet de tester que des égalités de valeurs entières (`byte`, `short`, `int` ou `long`) ou de type caractère (`char`). Elle **ne peut** donc **pas** être utilisée pour :

- Tester des valeurs réelles (`float` ou `double`) ou des mots (`String`).
- Rechercher si la valeur est plus grande, plus petite ou différente d'une certaine étiquette.

Par contre, l'instruction `if-else` peut être employée dans tous les cas en testant tout type de variable, selon toute condition.

Remarquons cependant que :

- Si une condition parmi d'autres conditions envisagées a une plus grande probabilité d'être satisfaite, celle-ci doit être placée en premier test dans une structure `if else`, de façon à éviter à l'ordinateur d'effectuer de trop nombreux tests inutiles.
- Si toutes les conditions ont une probabilité voisine ou équivalente d'être réalisées, la structure `switch` est plus efficace. Elle ne demande qu'une seule évaluation, alors que, dans les instructions `if-else` imbriquées, chaque condition doit être évaluée.

Résumé

L'instruction `if else` (traduction : si, sinon) permet de programmer des choix. De façon générale, l'instruction `if else` s'écrit :

```
if     (condition)
// si la condition est vraie
{          // faire
    plusieurs instructions ;
}        // fait
else     // sinon
{        //faire
    plusieurs instructions ;
}        //fait
```

Ou encore

```
if (condition) une seule instruction ;
else           une seule instruction ;
```

- Si la condition située après le mot-clé `if` (placée obligatoirement entre parenthèses) est vraie, alors les instructions placées dans le bloc défini par les accolades ouvrante et fermante immédiatement après sont exécutées.
- Si la condition est fausse, alors les instructions définies dans le bloc situé après le mot-clé `else` sont exécutées.

De cette façon, un seul des deux blocs est exécuté, selon que la condition est vérifiée ou non. De plus, cette condition fait intervenir des :

- Opérateurs relationnels :

Opérateur	Signification pour des valeurs numériques	Signification pour des valeurs de type caractère
= =	égal	identique
<	inférieur strictement	plus petit dans l'ordre alphabétique
<=	inférieur ou égal	plus petit ou identique dans l'ordre alphabétique
>	supérieur strictement	plus grand dans l'ordre alphabétique
>=	supérieur ou égal	plus grand ou identique dans l'ordre alphabétique
! =	différent	différent

• Opérateurs logiques :

Opérateur	Signification
!	NON logique
&&	ET logique
||	OU logique

Lorsque plusieurs instructions `if-else` sont imbriquées les unes dans les autres, un `else` se rapporte toujours au dernier bloc `if` rencontré auquel un `else` n'a pas encore été attribué.

L'instruction `switch` (traduction : selon ou suivant) permet de programmer des choix multiples. Elle a pour syntaxe :

```
switch(valeur)          // le type de la variable est char ou int
{
   case étiquette :     // suite d'instructions
          break ;       // facultatif, pour sortir du bloc switch
   case étiquette :     // suite d'instructions
          break ;       // facultatif, pour sortir du bloc switch
   default        :     // suite d'instructions
}
```

La variable `valeur` est évaluée. Suivant cette évaluation, le programme recherche l'`étiquette` correspondant à la valeur évaluée et définie à partir des instructions `case étiquette`.

• Si le programme trouve une étiquette correspondant au contenu de la variable `valeur`, il exécute la ou les instructions qui suivent l'étiquette, jusqu'à rencontrer le mot-clé `break`.
• S'il n'existe pas d'étiquette correspondant à `valeur`, alors le programme exécute les instructions de l'étiquette `default`.

L'instruction `if-else` est utilisée lorsque l'une des conditions envisagées a une grande probabilité d'être satisfaite. Si toutes les conditions ont une probabilité d'être réalisées, on utilise plutôt la structure `switch`.

Exercices

Comprendre les niveaux d'imbrication

3.1 Exécutez à la main (c'est-à-dire ligne par ligne) ce programme. Pour cela, vous supposerez que la valeur saisie au clavier soit 4. Quel est le résultat affiché ?

```
public class Racine
{
  public static void main (String [] parametre)
  {
    double  x, r ;
```

```
    System.out.print("Entrer un chiffre :") ;
    x =  Lire.d() ;
    if (x > = 0)
    {
      r = Math.sqrt(x) ;
    }
    else
    {
      r = Math.sqrt(-x) ;
    }
    System.out.print("Pour "+x+" Le resultat est: "+r) ;
  } // Fin du main ()
} // Fin de la Class Racine
```

Même question en supposant la valeur saisie égale à – 9.

Construire une arborescence de choix

3.2 Reprenez et modifiez le programme `Maximum` donné dans ce chapitre, de façon qu'il affiche un message lorsque les deux valeurs saisies au clavier sont égales.

3.3 Représentez graphiquement les choix arborescents suivants :

```
if (Condition 1)
    {
        if (Condition 2)
        {
            if (Condition 3)
            {
                 instruction A
            }
        }
        else
        {
            instruction B
        }
    }
    else
    {
        instruction C
    }
```

3.4 Écrivez un programme qui résolve les équations du second degré à l'aide de structures `if-else` imbriquées.

Soit l'équation $ax^2 + bx + c = 0$, où `a`, `b`, et `c` représentent les trois coefficients entiers de l'équation. Pour trouver les solutions réelles `x`, si elles existent :

a. Établissez l'arbre des choix associés :

```
1.     a = 0
    1.1.     b = 0
        1.1.1.    c = 0         tout réel est solution
```

```
        1.1.2.    c != 0        pas de solution
    1.2.    b != 0               une seule solution : x = - c / b ;
2.     a != 0
    2.1.    b² - 4ac >= 0     deux solutions :
            x1 = - b + Math.sqrt(b * b - 4 * a * c) / 2 * a ;
            x2 = - b - Math.sqrt(b * b - 4 * a * c) / 2 * a ;
    2.2.    b² - 4ac < 0        pas de solution dans les réels
```

b. Déterminez les différentes variables à déclarer.

c. À partir de l'arbre des choix, écrivez les instructions `if-else` suivies du test correspondant.

d. Placez dans chaque bloc `if` ou `else` les instructions de calcul et d'affichage appropriées.

e. Placez l'ensemble de ces instructions dans une fonction `main()` et une classe portant le nom `SecondDegre`.

Manipuler les choix multiples, gérer les caractères

3.5 En utilisant la structure `switch`, écrire un programme qui simule une machine à calculer dont les opérations soient l'addition (+), la soustraction (–), la multiplication (*) et la division (/).

a. En cours d'exécution, le programme demande à l'utilisateur d'entrer deux valeurs numériques puis le caractère correspondant à l'opération à effectuer. Suivant le caractère entré (+-*/) le programme affiche l'opération effectuée, ainsi que le résultat.

L'exécution du programme peut, par exemple, avoir l'allure suivante (les valeurs grisées sont celles saisies par l'utilisateur) :

```
Entrez la premiere valeur : 2
Entrez la seconde valeur : 3
Type de l'operation  (+, -, *, /) : *
Cette operation a pour resultat : 2 * 3 = 6
```

b. Après avoir écrit et exécuté le programme avec différentes valeurs, saisissez dans cet ordre les valeurs suivantes : `2`, `0` puis `/`. Que se passe-t-il ? Pourquoi ?

c. Modifiez le programme de façon à ne plus rencontrer cette situation en cours d'exécution.

Le projet « Gestion d'un compte bancaire »

Accéder à un menu suivant l'option choisie

L'objectif est d'améliorer le programme réalisé à la fin du chapitre 2, « Communiquer une information », afin d'afficher chaque menu en fonction de l'option choisie par l'utilisateur.

a. Après l'affichage du menu principal, le programme teste la valeur entrée par l'utilisateur et affiche l'option correspondante. Sachant que toutes les options du menu

principal ont une probabilité voisine ou équivalente d'être réalisées, quelle est la structure de test la plus appropriée ?

b. Modifiez le programme en fonction de la structure de test choisi, et placez les instructions d'affichage et de saisie dans les options correspondantes.

c. Pour l'`option 1`, testez le type du compte afin de saisir le taux d'épargne.

d. Pour l'`option 2`, demandez au programme de vérifier que le numéro du compte saisi par l'utilisateur existe, de façon à :

- Afficher le numéro du compte, le type, la valeur initiale et son taux dans le cas d'un compte d'épargne, si le compte existe.
- Afficher un message indiquant que le numéro du compte n'est pas valide, si le compte n'existe pas.

4

Faire des répétitions

La notion de répétition est une des notions fondamentales de la programmation. En effet, beaucoup de traitements informatiques sont répétitifs. Par exemple, la création d'un agenda électronique nécessite de saisir un nom, un prénom et un numéro de téléphone autant de fois qu'il y a de personnes dans l'agenda.

Dans de tels cas, la solution n'est pas d'écrire un programme qui comporte autant d'instructions de saisie qu'il y a de personnes mais de faire répéter par le programme le jeu d'instructions nécessaires à la saisie d'une seule personne. Pour ce faire, le programmeur utilise des instructions spécifiques, appelées structures de répétition, ou **boucles**, qui permettent de déterminer la ou les instructions à répéter.

Dans ce chapitre, nous abordons la notion de répétition à partir d'un exemple imagé (*« Combien de sucre dans votre café »*).

Nous étudions ensuite les différentes structures de boucles proposées par le langage Java (*sections « La boucle* `do...while`*», « La boucle* `while`*» et « La boucle* `for`*»*). Pour chacune de ces structures, nous présentons et analysons un exemple afin d'examiner les différentes techniques de programmation associées aux structures répétitives.

Combien de sucre dans votre café ?

Pour bien comprendre la notion de répétition ou de boucle, nous allons améliorer l'algorithme du café chaud sucré, de sorte que le programme demande à l'utilisateur de prendre un morceau de sucre autant de fois qu'il le souhaite. Pour cela, nous reprenons uniquement le bloc d'instructions `II Préparer le sucre` (*voir, au chapitre 3, « Faire des choix », la section « L'algorithme du café chaud, sucré ou non »*).

Instructions	Bloc d'instructions
`Si (café sucré)`	

`1. Prendre une petite cuillère.`
`2. Poser la petite cuillère dans la tasse.`
`3. Prendre un morceau de sucre.`
`4. Poser le sucre dans la tasse.`
	II . Préparer le sucre

L'exécution du bloc d'instructions `II Préparer le sucre` nous permet de mettre un seul morceau de sucre dans la tasse. Si nous désirons mettre plus de sucre, nous devons exécuter les instructions `3` et `4` autant de fois que nous souhaitons de morceaux de sucre. Remarquons que, dans ce bloc, les instructions `1` et `2` ne sont pas à répéter, sous peine d'avoir autant de petites cuillères que de morceaux de sucre dans la tasse. La marche à suivre devient dès lors :

```
Prendre une petite cuillère.
Poser la petite cuillère dans la tasse.
Début répéter :
        1.    Prendre un morceau de sucre.
        2.    Poser le sucre dans la tasse.
        3.    Poser la question : "Souhaitez-vous un autre morceau de sucre ?"
        4.    Attendre la réponse.
Tant que la réponse est OUI, retourner à Début répéter.
```

Analysons les résultats possibles de cette nouvelle marche à suivre :

- Dans tous les cas, nous prenons et posons une petite cuillère.
- Ensuite, nous entrons sans condition dans une structure de répétition.
- Nous prenons et posons un morceau de sucre, quelle que soit la suite des opérations. De cette façon, si nous sortons de la boucle, le café est quand même sucré.
- Puis le programme nous demande si nous souhaitons à nouveau un morceau de sucre.
- Si notre réponse est `OUI`, le programme retourne au début de la structure répétitive, place le sucre dans la tasse et demande de nouveau si nous souhaitons du sucre, etc.
- Si la réponse est négative, la répétition s'arrête, ainsi que la marche à suivre.

Pour écrire une boucle, nous constatons que :

- Il est nécessaire de déterminer où se trouve le début de la boucle et où se situe la fin (`Début répéter` et `Tant que` pour notre exemple).
- La sortie de la structure répétitive est soumise à la réalisation ou non d'une condition (`la réponse` fournie est-elle affirmative ou non ?).
- Le résultat du test de sortie de boucle est modifiable par une instruction placée à l'intérieur de la boucle (la valeur de la réponse est modifiée par l'instruction `4. Attendre la réponse`).

Dans le langage informatique, la construction d'une répétition ou boucle suit le même modèle.

Dans le langage Java, il existe trois types de boucles, qui sont décrites par les constructions suivantes :

Type de boucle	Signification
`do...while`	Faire... tant que
`while`	Tant que
`for`	Pour

Dans la suite de ce chapitre, nous allons, pour chacune de ces boucles :

- Étudier la syntaxe.
- Analyser les principes de fonctionnement.
- Donner un exemple qui introduise un concept fondamental de la programmation, à savoir le compteur de boucle, l'accumulation de valeurs ou la recherche d'une donnée parmi un ensemble d'informations.

La boucle `do...while`

La boucle `do…while` est une structure répétitive, dont les instructions sont exécutées avant même de tester la condition d'exécution de la boucle. Pour construire une telle structure, il est nécessaire de suivre les règles de syntaxe décrites ci-après.

Syntaxe

La boucle `do…while` se traduit par les termes `faire... tant que`. Cette structure s'écrit de deux façons différentes en fonction du nombre d'instructions qu'elle comprend.

Dans le cas où une seule instruction doit être répétée, la boucle s'écrit de la façon suivante :

```
do
    une seule instruction ;
while (expression conditionnelle) ;
```

Si la boucle est composée d'au moins deux instructions, celles-ci sont encadrées par des accolades, ouvrante et fermante, de façon à déterminer où commence et se termine la boucle.

```
do {
    plusieurs instructions ;
} while (expression conditionnelle) ;
```

Principes de fonctionnement

Ainsi décrite, la boucle `do…while` s'exécute selon les principes suivants :

- Les instructions situées à l'intérieur de la boucle sont exécutées tant que l'expression conditionnelle placée entre parenthèses`()` est vraie.

- Les instructions sont exécutées au moins une fois, puisque l'expression conditionnelle est examinée en fin de boucle, après exécution des instructions.
- Si la condition mentionnée entre parenthèses reste toujours vraie, les instructions de la boucle sont répétées à l'infini. On dit que le programme « boucle ».
- Une instruction modifiant le résultat du test de sortie de boucle est placée à l'intérieur de la boucle, de façon à stopper les répétitions au moment souhaité.
- Remarquons qu'un point-virgule est placé à la fin de l'instruction `while (expression);`.

Un distributeur automatique de café

L'objectif de cet exemple est double : apprendre à construire une boucle `do…while` et étudier comment compter et accumuler des valeurs.

Le comptage des valeurs, quelles qu'elles soient, est une technique très utilisée en informatique. Il existe deux façons de compter :

- Le **comptage** d'un certain nombre de valeurs. Par exemple, le programme compte le nombre de notes d'un étudiant.
- L'**accumulation** de valeurs. Le programme calcule la somme des notes d'un étudiant (les notes sont accumulées).

Le calcul de la moyenne des notes d'un étudiant s'effectue en divisant l'accumulation des notes par le nombre (comptage) de notes obtenues.

Pour bien comprendre ces différentes techniques, nous allons écrire un programme dont l'objectif est de simuler de façon simplifiée un distributeur automatique de café.

Cahier des charges

Pour obtenir un café, l'utilisateur introduit un certain nombre de pièces de monnaie dans le distributeur. Pour simplifier, nous supposons que l'appareil n'accepte que les pièces de 1, 2 et 5 F. Lorsqu'une pièce est introduite, le distributeur affiche la valeur totale engagée, ainsi que le nombre de pièces par catégorie (nombre de pièces de 1 F, 2 F et 5 F). La machine prépare un café dès que la somme totale introduite vaut ou dépasse le prix du café. Nous prenons pour hypothèse que le prix d'un café soit de 3 F. La machine rend la monnaie, s'il y a lieu.

Après lecture et analyse du cahier des charges, nous remarquons que la démarche se déroule en trois temps.

1. Introduction une à une des pièces dans le distributeur.

2. À chaque pièce fournie, calcul et affichage :

   a. Du nombre de pièces de 1 F, 2 F et 5 F.

   b. De la somme engagée.

3. Y a-t-il suffisamment d'argent ?

   a. Non, alors retourner en 1.

   b. Oui, alors préparer le café et rendre la monnaie.

Pour écrire le programme, nous allons nous attacher à résoudre, dans l'ordre, chacun de ces points.

1. Construire la boucle et introduire les pièces.

Les points 1 et 3.a décrivent la structure de la boucle. L'introduction des pièces dans le distributeur est une opération répétitive, qui s'arrête lorsque l'utilisateur a placé suffisamment d'argent dans le distributeur, c'est-à-dire lorsque le montant total engagé vaut ou dépasse la somme de 3 F. Par conséquent, l'allure générale de la structure répétitive est la suivante :

```
Début répéter
    Entrer une pièce de monnaie
    Compter la somme engagée
Tant que la somme engagée ne dépasse pas 3 F, retourner à Début répéter.
```

En langage Java, cette structure est traduite en reprenant la syntaxe de la boucle `do…while`, c'est-à-dire par :

```
do // Début de boucle
{
    // Entrer les pièces de monnaie
    // Compter la somme engagée
}
while (somme engagée < 3 F); // Fin de boucle
```

De cette façon, la boucle est exécutée tant que la somme engagée est inférieure à 3 F. Dès que cette somme vaut ou dépasse 3 F, la condition `somme engagée < 3 F` n'est plus vérifiée, et le programme sort de la boucle.

Ensuite, pour simuler l'introduction des pièces de monnaie dans le distributeur, le programme demande à l'utilisateur de saisir au clavier la valeur de chaque pièce entrée. Nous écrivons donc :

```
System.out.println("valeur de la piece entree :");
pièce = Lire.b();
```

✔ Pour plus d'informations voir le chapitre 2, *« Communiquer une information ».*

2. Compter le nombre de pièces et la somme totale engagée.

Pour compter le nombre de pièces de 1 F, 2 F et 5 F, le programme doit pouvoir distinguer les différentes pièces introduites. Pour cela, nous déclarons autant de variables qu'il y a de catégories de pièces, soit :

```
byte nbPièce1F = 0, nbPièce2F= 0, nbPièce5F=0, pièce, totalReçu = 0;
```

Les variables dont le nom commence par `nb` représentent le nombre de pièces pour chacune des catégories. La variable `pièce` désigne, quant à elle, la valeur de la pièce saisie au clavier. Enfin, la variable `totalReçu` représente la somme totale engagée en cours d'exécution de la boucle. Ces variables sont déclarées de type `byte` (leur valeur ne dépasse jamais 127).

a. Pour compter séparément les pièces de 1 F, de 2 F et de 5 F, la meilleure méthode consiste à placer dans la boucle `do…while` une structure `switch` distinguant trois cas :

```
switch (pièce)
  {
    case 1 :
      // Compter les pièces de 1 F
      break;
    case 2 :
      // Compter les pièces de 2 F
      break;
    case 5 :
      // Compter les pièces de 5 F
      break;
    default :
      System.out.println ("Piece impossible");
  }
```

Suivant la valeur de la pièce engagée, le programme compte le nombre de pièces, pour chacune des catégories en utilisant une instruction du type :

```
a = a + 1;
```

où `a` représente l'objet à compter. Si la variable `a` est initialisée à `0`, la nouvelle valeur de `a`, après affectation, vaut `1`.

✔ Pour plus d'informations voir, au chapitre 1, *« Stocker une information »,* la section *« Quelques confusions à éviter ».*

Placé dans une structure répétitive, le nombre d'objets représentés par `a` augmente de 1 à chaque tour de boucle. En informatique, on dit que `a` est **incrémenté** de 1. Pour compter le nombre de pièces de 1 F, 2 F et 5 F, il suffit de remplacer la variable `a` par `nbPièce1F`, `nbPièce2F` ou `nbPièce5F`. Nous obtenons ainsi, pour chaque catégorie de pièces, les instructions suivantes :

```
nbPièce1F = nbPièce1F + 1;
nbPièce2F = nbPièce2F + 1;
nbPièce5F = nbPièce5F + 1;
```

b. Ces instructions sont ensuite placées dans les étiquettes `1`, `2` et `5` de la structure `switch`.

✔ Pour mieux comprendre l'évolution de la valeur de ces variables, reportez-vous à la section *« Résultat de l'exécution ».*

Pour calculer la somme engagée à chaque pièce introduite, la technique est légèrement différente de la précédente. En effet, la somme engagée doit être augmentée non plus du nombre de pièces introduites mais de la **valeur** de la pièce introduite. L'incrément n'est plus de 1 mais de la valeur de la pièce. Comme la variable `pièce` représente la valeur de la pièce, l'instruction d'**accumulation** est la suivante :

```
totalReçu = totalReçu + pièce;
```

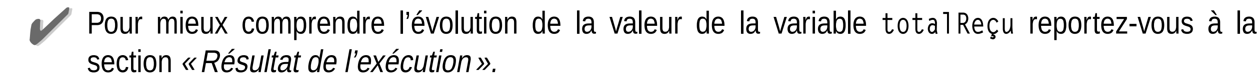
Ainsi, la variable `totalReçu`, initialisée à zéro, augmente progressivement de la valeur de chaque pièce engagée, par accumulation de la valeur précédente de `totalReçu` avec la valeur de la pièce entrée.

Ce calcul est réalisé quelle que soit la valeur de la pièce. Par conséquent, cette instruction est placée en dehors de la structure `switch`, mais, à l'intérieur de la boucle. Le montant total engagé est modifié chaque fois qu'une nouvelle pièce de 1 F, 2 F ou 5 F est introduite.

Pour éviter d'accumuler dans `totalReçu` la valeur d'une pièce non autorisée, nous devons modifier la valeur de la pièce dans l'étiquette `default` de la structure `switch` par l'instruction :

```
default :
       pièce = 0;
       System.out.println ("Piece impossible");
```

Lorsqu'une mauvaise pièce est introduite, la variable `pièce` prend la valeur `0`. De cette façon, l'instruction d'accumulation est réalisée, quelle que soit la valeur de la pièce, puisque la variable `totalReçu` n'est pas modifiée par l'accumulation d'une pièce valant 0 F.


✔ Pour mieux comprendre l'évolution de la valeur de la variable `totalReçu` reportez-vous à la section *« Résultat de l'exécution »*.

Une fois le nombre de pièces compté et le montant total calculé, le programme affiche les différentes valeurs à l'aide des instructions suivantes :

```
System.out.println("Vous avez entre  : ");
System.out.println("    " + nbPièce1F + " piece(s) de 1 F");
System.out.println("    " + nbPièce2F + " piece(s) de 2 F");
System.out.println("    " + nbPièce5F + " piece(s) de 5 F");
System.out.println("Soit au total :  " + totalReçu + " F");
```

L'ensemble de ces instructions est placé avant le test de sortie de boucle, puisque les valeurs calculées sont affichées chaque fois que l'utilisateur entre une pièce.

3. Y a-t-il suffisamment d'argent ?

a. Non, alors retourner en 1.

Il s'agit de déterminer la condition de sortie ou non de la boucle. Cette opération est décrite au point 1.

Remarquons, cependant, que grâce à l'instruction d'accumulation

```
totalReçu = totalReçu + pièce;
```

la valeur de la variable `totalReçu` est augmentée à chaque tour de boucle. Par conséquent, le résultat de la condition de sortie de boucle (`totalReçu < 3`) ne reste pas toujours vrai. Le programme peut sortir de la boucle.

b. Oui, alors préparer le café et rendre la monnaie.

Lorsque l'utilisateur a entré suffisamment de pièces de monnaie, le programme affiche un message qui annonce que le café est prêt, à l'aide de l'instruction :

```
System.out.println("Je vous verse 1 cafe ");
```

Pour détecter un trop-perçu, le programme teste si `totalReçu` dépasse la valeur du prix du café. Si tel est le cas, il calcule la monnaie à rendre et affiche un message en conséquence. Ces actions sont réalisées par les instructions :

```
if (totalReçu > 3)
System.out.println("et vous rends : " + (totalReçu-3) + " F ");
```

Exemple : code source

Pour obtenir un programme à part entière, l'ensemble des instructions développées au cours de la section précédente est à placer dans une fonction `main()` et une classe, comme ci-dessous :

```
public class CompteurMonnaie
{
 public static void main(String [] arg)
 {
  byte nbPièce1F = 0, nbPièce2F= 0, nbPièce5F=0, pièce;
  byte totalReçu = 0;
  System.out.println("Pour obtenir un cafe, entrez au moins 3 F");
  System.out.println("Je rends la  monnaie ");
  do
  {
   System.out.println("valeur de la piece entree :");
   pièce = Lire.b();
   switch (pièce)
   {
     case 1 :
       nbPièce1F = nbPièce1F + 1;
       break;
     case 2 :
       nbPièce2F = nbPièce2F + 1;
       break;
     case 5 :
       nbPièce5F = nbPièce5F + 1;
       break;
     default :
       pièce = 0;
       System.out.println ("Piece impossible");
   }
   totalReçu = totalReçu + pièce;
   System.out.println("Vous avez entre  : );
   System.out.println("    " + nbPièce1F + " piece(s) de 1 F");
   System.out.println("    " + nbPièce2F + " piece(s) de 2 F");
   System.out.println("    " + nbPièce5F + " piece(s) de 5 F");
   System.out.println("Soit au total :  " + totalReçu + " F");
  } while (totalReçu < 3);
  System.out.println("Je vous verse 1 cafe ");
  if (totalReçu > 3)
```

```
      System.out.println("et vous rends : " + (totalReçu-3) + " F ");
    }
  }
```

Résultat de l'exécution

À l'écran	pièce	nb Pièce1F	nb Pièce2F	nb Pièce5F	total Reçu
Valeurs initiales au début de l'exécution	-	0	0	0	0

`Valeur de la piece entree : 2`
`Vous avez entre :`
`0 piece(s) de 1 F`
`1 piece(s) de 2 F`
`0 piece(s) de 5 F`
`Soit au total : 2 F`
	2	0	1
			car
			0 + 1
				0	2
					car
					0 + 2

`Valeur de la piece entree : 10`
`Piece impossible`
`Vous avez entre :`
`0 piece(s) de 1 F`
`1 piece(s) de 2 F`
`0 piece(s) de 5 F`
`Soit au total : 2 F`
	10
	
	0
		0	1	0	2
					car
					2 + 0

`Valeur de la piece entree : 5`
`Vous avez entre :`
`0 piece(s) de 1 F`
`1 piece(s) de 2 F`
`1 piece(s) de 5 F`
`Soit au total : 7 F`
	5	0	1	1
				car 0 +
				1 = 1
					7
					car
					2 + 5

`Je vous verse 1 cafe` `et vous rends : 4 F`	5	0	1	1	7

La boucle `while`

Le langage Java propose une autre structure répétitive, analogue à la boucle `do…while`, mais dont la décision de poursuivre la répétition s'effectue en début de boucle. Il s'agit de la boucle `while`.

Syntaxe

La boucle `while` s'écrit de deux façons différentes, en fonction du nombre d'instructions qu'elle comprend.

Dans le cas où une seule instruction doit être répétée, la boucle s'écrit :

```
while (expression conditionnelle)
    une seule instruction ;
```

Si la boucle est composée d'au moins deux instructions, celles-ci sont encadrées par des accolades, ouvrante et fermante, de façon à déterminer où débute et se termine la boucle.

```
while (expression conditionelle)
{
    plusieurs instructions ;
}
```

Principes de fonctionnement

Le terme `while` se traduit par `tant que`. La structure répétitive s'exécute selon les principes suivants :

- Tant que l'expression à l'intérieur des parenthèses reste vraie, la ou les instructions composant la boucle sont exécutées.
- Le programme sort de la boucle dès que l'expression à l'intérieur des parenthèses devient fausse.
- Une instruction est placée à l'intérieur de la boucle pour modifier le résultat du test à l'entrée de la boucle, de façon à stopper les répétitions.
- Si l'expression à l'intérieur des parenthèses est fausse dès le départ, les instructions ne sont jamais exécutées.
- Observons qu'à l'inverse de la boucle `do…while`, il n'y a pas de point-virgule à la fin de l'instruction `while (expression)`.

Saisir un nombre entier au clavier

L'objectif de cet exemple est d'apprendre à écrire une boucle `while` et de comprendre comment réaliser la saisie d'un entier au clavier telle qu'elle est réalisée dans le programme `Lire.java`.

Nous avons déjà remarqué (*voir, au chapitre 2, « Communiquer une information », la section « La saisie de données »*) que la fonction `System.in.read()` ne permettait de saisir qu'un seul caractère à la fois au clavier. Pour saisir un nombre composé de plusieurs chiffres ou un mot constitué de plusieurs caractères, nous devons faire appel à la fonction `System.in.read()` autant de fois qu'il y a de caractères à saisir.

Cette saisie de caractères est donc une opération répétitive, qui doit s'arrêter lorsque la valeur numérique ou le mot est entièrement entré. L'ordinateur n'est pas à même de déterminer quand la saisie est terminée. L'utilisateur confirme qu'il a fini d'entrer des valeurs en appuyant sur une touche caractéristique du clavier. Cette touche, utilisée pour passer à la ligne dans les logiciels de traitement de texte, est communément appelée la touche « `Entrée` ».

Notre but étant de saisir une valeur numérique entière, nous devons traduire l'ensemble des caractères saisis, de façon à les stocker non plus dans un `String` mais dans une variable de type `int`. Si cette traduction n'est pas réalisée, il n'est pas possible d'additionner ou de diviser les caractères lus à la manière des valeurs numériques. Par exemple, le fait d'additionner la suite de caractères 123 avec la valeur 4 a pour résultat 1234. Par

contre, après traduction des caractères en valeur numérique, la même opération donne pour résultat 127.

Cahier des charges

Nous venons de l'observer, pour confirmer que nous n'avons plus de caractère à saisir, nous devons appuyer sur la touche « Entrée » du clavier. Pour saisir une valeur numérique entière, la liste des opérations s'exprime sous la forme de la structure répétitive suivante :

1. Tant que le caractère saisi n'est pas le caractère « `Entrée` » :

   a. Lire un caractère.

   b. Stocker le caractère lu dans un mot.

   Retourner en **1.**

2. Tous les caractères étant saisis, les traduire en un nombre entier.

Pour écrire le programme en langage Java, reprenons cette marche à suivre point par point.

1. La boucle `tant que` est traduite en Java par la construction suivante :

```
while (C != '\n')
{
 // Lire un caractère au clavier
 // Stocker le caractère dans un mot
}
```

En Java, le caractère « `Entrée` » est symbolisé par le caractère `'\n'` sur des ordinateurs de type Unix ou Macintosh. Sur un PC, la touche « `Entrée` » correspond à la série de caractères `'\r'` et `'\n'`. Afin de rendre compatible le programme avec tous les ordinateurs, nous allons tester la condition de sortie de boucle sur le caractère `'\n'`, puisque celui-ci est commun à tous les mondes, qu'ils soient Unix, Macintosh ou PC. De cette façon, en écrivant `while (C != '\n')`, où `C` représente le caractère lu, nous exprimons en langage informatique la phrase : tant que le caractère saisi n'est pas le caractère « `Entrée` ».

La première fois que le programme entre dans la boucle, aucun caractère n'a encore été saisi. Il est donc nécessaire d'initialiser la variable `C` à un caractère différent de `'\n'`, de façon à assurer que la condition d'entrée dans la boucle soit au moins vérifiée la première fois. Pour cela, nous déclarons `C` en début de programme, de la façon suivante :

```
char C ='\0';
```

Par cette instruction, nous initialisons la variable `C` au caractère nul (`'\0'`). Nous aurions pu l'initialiser à tout autre caractère à condition que celui-ci fût différent de `'\n'`. Le choix du caractère nul n'est ici réalisé que parce que, en général, les variables de type entier ou réel sont initialisées à `0` ou `0.0`. En Java, le caractère `'\0'` est l'équivalent de la valeur numérique nulle.

a. Pour lire un caractère au clavier, l'instruction est la suivante :

```
C = (char) System.in.read() ;
```

La fonction `System.in.read()` attend que l'utilisateur appuie sur une touche du clavier. Cela fait, elle retourne en résultat la valeur entière correspondant au caractère associé à la touche du clavier. Pour traduire cette valeur entière en code caractère, il est nécessaire de placer le cast `(char)` devant la fonction. De cette façon, la variable `C` contient le code Unicode du caractère saisi.

b. Stocker le caractère lu dans un mot.

L'objectif est de lire plusieurs caractères d'affilée. Nous devons donc stocker dans une variable de type `String` chaque caractère au fur et à mesure de la saisie (*voir, au chapitre 7, « Les classes et les objets », la section « La classe* `String`*, une approche vers la notion d'objet »*). Grâce au type `String`, plusieurs caractères peuvent être stockés sous un même nom de variable. La méthode consiste à accumuler dans une variable les valeurs lues, en utilisant l'instruction :

```
tmp = tmp + C;
```

Cette instruction permet d'accumuler les valeurs saisies en les plaçant les unes derrière les autres dans la variable `tmp`. En effet, lorsque deux caractères sont additionnés, ceux-ci sont placés dans la variable l'un après l'autre dans l'ordre d'exécution de l'opération. L'addition du caractère `'e'` et du caractère `'t'` a pour résultat le mot `et`. Dans le jargon informatique, l'addition de caractères est aussi appelée la **concaténation** de caractères.

En début de programme, la variable `tmp` ne doit pas contenir de caractère. Cela vient du fait que, la première fois qu'un caractère lu est placé dans la variable `tmp`, il doit correspondre au tout premier caractère du mot stocké dans la variable `tmp`. C'est pourquoi, la variable `tmp` doit être déclarée de la façon suivante (`""` correspondant à un mot vide de caractère) :

```
String tmp = "";
```

Lorsque, au final, l'utilisateur appuie sur la touche « Entrée » pour valider la fin de la saisie, le programme (sur PC) reçoit la suite de caractères `'\r'` et `'\n'`. La variable `tmp` contient en définitive la suite des caractères saisis, plus les caractères `'\r'` et `'\n'`. Or, nous souhaitons transformer cette suite de caractères en valeur numérique. Pour cela, nous devons éliminer les caractères `'\r'` et `'\n'`, qui empêchent cette transformation.

✔ Pour plus d'informations, reportez-vous au paragraphe *« 2. Traduire les caractères en un nombre entier »*, un peu plus loin dans ce chapitre.

L'accumulation des caractères ne se réalise donc qu'à la condition que le caractère saisi ne soit égal ni à `'\r'`, ni à `'\n'`.

Pour résumer, la boucle s'écrit :

```
String tmp = "";
char C ='\0';
while (C != '\n')
```

```
{
 C = (char) System.in.read() ;
 if (C != '\r' && C != '\n') tmp = tmp + C;
}
```

Pour mieux comprendre en pratique le déroulement de cette boucle, examinons l'évolution des variables à partir d'un exemple. Nous supposons que l'utilisateur entre les caractères `2`, `8` et « `Entrée` ».

	c	tmp	Explication
`String tmp = "";`	\0	-	Initialisation
`char C ='\0';`	\0	""	Initialisation
`while (C != '\n')` `{`	\0	""	C étant initialisé au caractère '\0', `C` est différent du caractère `'\n'`. La condition placée entre `()` est vérifiée. Le programme entre dans la boucle.
`C = (char)System.in.read();`	2	""	Le programme attend la saisie d'une valeur au clavier. Nous supposons que le caractère saisi soit `2`.
`if (C != '\r' && C != '\n')` `  tmp = tmp + C;`	2	2	Le caractère `C` étant différent de `'\r'`, la concaténation est exécutée. La variable `tmp` étant initialisée à la chaîne vide (`""`), l'opération `"" + '2'` stocke le caracère `2` en première position dans la variable `tmp`.
`}`	2	2	Fin de boucle. Le programme retourne en début de boucle.
`while (C != '\n')` `{`	2	2	La variable `C` contient la valeur 2. `C` est donc différent du caractère `'\n'`. La condition placée entre `()` est vérifiée. Le programme entre dans la boucle.
`C = (char)System.in.read();`	8	2	Nous entrons le caractère `8`.
`if (C != '\r' && C != '\n')` `  tmp = tmp + C;`	8	28	Le caractère `C` étant différent de `'\r'`, l'opération `'2' + '8'` est exécutée et stocke le mot `28` dans la variable `tmp`.
`}`	8	28	Fin de boucle. Le programme retourne en début de boucle.
`while (C != '\n')` `{`	8	28	La variable `C` contient le caractère 8. `C` est donc différent du caractère `'\n'`. La condition placée entre `()` est vérifiée. Le programme entre dans la boucle.
`C=(char)System.in.read();`	\r	28	Nous appuyons sur la touche « Entrée ». Sur PC, le premier caractère entré est `'\r'`.
`if (C != '\r' && C != '\n')` `  tmp = tmp+C;`	\r	28	C vaut `'\r'`. La condition n'étant pas vérifiée, il n'y a pas accumulation du caractère dans `tmp`.
`}`	\r	28	Fin de boucle. Le programme retourne en début de boucle.
`while (C != '\n')` `{`	\r	28	La variable `C` contient le caractère `\r`. `C` est donc différent du caractère `'\n'`. La condition placée entre `()` est vérifiée. Le programme entre dans la boucle

	c	tmp	Explication
`C = (char)System.in.read();`	\n	28	Le caractère suivant envoyé par la touche « Entrée » est '\n'.
`if (C != '\r' && C != '\n')` `tmp = tmp + C;`	\n	28	C vaut '\n'. La condition n'est pas vérifiée, et il n'y a pas accumulation du caractère dans tmp.
`}`	\n	28	Fin de boucle. Le programme retourne en début de boucle.
`while (C !='\n')` `{`	\n	28	La variable C contient le caractère \n. La condition placée entre () n'est plus vérifiée. Le programme sort de la boucle et passe à l'étape suivante.

2. Traduire les caractères en un nombre entier.

Pour traduire un ensemble de caractères en une valeur numérique, le langage Java propose un certain nombre de fonctions. Dans notre cas, il s'agit de traduire un mot en une valeur entière de type `int`. La fonction Java `Integer.parseInt()` permet une telle traduction. L'instruction est la suivante :

```
valeur = Integer.parseInt(tmp);
```

`valeur` est une variable déclarée de type `int`, et `tmp` est le mot qui contient les caractères à traduire. La variable `tmp` ne doit contenir que des caractères représentant des chiffres. Si tel n'est pas le cas, le programme s'arrête avec un message d'erreur à l'exécution. Par exemple, si l'utilisateur entre le mot `deux`, au lieux du caractère `2`, l'interpréteur Java affiche le message suivant :

```
java.lang.NumberFormatException : deux
at java.lang.Integer.parseInt (compiled Code)
```

Ce message indique que le format du nombre saisi ne correspond pas au format attendu par la fonction `Integer.parseInt()`. Nous aurions obtenu le même type d'erreur en stockant les caractères `'\r'` ou `'\n'` dans la variable `tmp`.

Pour connaître les autres fonctions permettant de traduire une chaîne de caractères en valeur numérique de type `double`, `float`, `long` ou `byte`, vous pouvez consulter, à l'aide d'un éditeur de texte, le fichier `Lire.java`, qui emploie toutes ces fonctions.

Pour finir, le programme affiche les différents résultats à l'aide de la fonction `System.out.println`. Cet affichage est réalisé à la fin du code source complet ci-dessous.

Exemple : code source complet

Pour obtenir un programme à part entière, l'ensemble des instructions développées au cours de la section précédente est à placer dans une fonction `main()` et une classe, comme ci-dessous :

```
public class LireUnEntier
{
  public static void main (String [] param) throws java.io.IOException
  {
```

```
    String tmp = "";
    char C= '\0';
    int valeur ;
    System.out.print("Entrez des chiffres et appuyez sur ");
    System.out.println("la touche Entree, pour valider la saisie : ");
    while (C != '\n')
    {
     C = (char) System.in.read() ;
     if (C != '\r' && C != '\n') tmp = tmp + C;
    }
    System.out.println("Vous avez entre : " + tmp);
    valeur = Integer.parseInt(tmp);
    System.out.println("C'est a dire : " + valeur + " en entier");
   } // Fin du main ()
 } // Fin de la Class LireUnEntier
```

Résultat de l'exécution

Les valeurs grisées correspondent aux valeurs saisies par l'utilisateur. Suivant les valeurs saisies, le programme donne un résultat différent.

Exécution 1

```
Entrez des chiffres et appuyez sur la touche Entree, pour valider la
saisie : 28
Vous avez entre : 28
C'est a dire : 28 en entier
```

La première valeur `28` affichée est un mot. L'addition de cette valeur avec le nombre `4` a pour résultat `284`. La deuxième valeur affichée est un nombre, et la même addition a pour résultat `32`.

Exécution 2

```
Entrez des chiffres et appuyez sur la touche Entree, pour valider la
saisie : trois
java.lang.NumberFormatException : trois
at java.lang.Integer.parseInt (compiled Code)
```

Le mot `trois` n'est pas un nombre mais un mot sans signification particulière pour l'ordinateur ; l'interpréteur Java ne peut traduire ce mot en un nombre entier.

Exécution 3

```
Entrez des chiffres et appuyez sur la touche Entree, pour valider la
saisie : 2.5
java.lang.NumberFormatException : 2.5
at java.lang.Integer.parseInt (compiled Code)
```

Le mot `2.5` n'a pas le format d'un nombre entier mais d'un nombre réel. La fonction `Integer.parseInt()` ne peut le traduire en un nombre entier.

La boucle `for`

L'instruction `for` permet d'écrire des boucles dont on connaît à l'avance le nombre d'itérations (de tours) à exécuter. Elle est équivalente à l'instruction `while` mais est plus simple à écrire.

Syntaxe

La boucle `for` s'écrit elle aussi de deux façons différentes en fonction du nombre d'instructions qu'elle comprend.

Dans le cas où une seule instruction doit être répétée, la boucle s'écrit :

```
for (initialisation; condition; incrément)
    une seule instruction;
```

Si la boucle est composée d'au moins deux instructions, celles-ci sont encadrées par deux accolades, ouvrante et fermante, de sorte à déterminer où débute et se termine la boucle.

```
for (initialisation; condition; incrément)
{
    plusieurs instructions;
}
```

Les termes `Initialisation`, `Condition` et `Incrément` sont des instructions séparées obligatoirement par des points-virgules (`;`). Ces instructions définissent une variable, ou indice, qui contrôle le bon déroulement de la boucle. Ainsi :

- `Initialisation` permet d'initialiser la variable représentant l'indice de la boucle (exemple : `i = 0`, `i` étant l'indice). Elle est la première instruction exécutée, à l'entrée de la boucle.
- `Condition` définit la condition à vérifier pour continuer à exécuter la boucle (exemple : `i < 10`). Elle est examinée avant chaque tour de boucle, y compris au premier tour de boucle.
- `Incrément` est l'instruction qui permet de modifier le résultat du test précédent en augmentant ou diminuant la valeur de la variable testée. L'incrément peut être augmenté ou diminué de *N*. *N* est appelé le **pas d'incrémentation** (exemple : `i = i + 2`). Cette instruction est exécutée à la fin de chaque tour de boucle.

Principes de fonctionnement

Les boucles `for` réalisent un nombre précis de boucles dépendant de la valeur initiale, de la valeur finale et du pas d'incrémentation. Voyons sur différents exemples comment ces boucles sont exécutées (tableau suivant).

Remarquons que :

- Le nombre de tours est identique dans chacune de ces boucles, malgré une définition différente pour chacune des instructions de contrôle.
- L'écriture de l'instruction `Incrément`, qui augmente ou diminue de `1` la variable de contrôle de la boucle, peut être simplifiée. En effet, par convention, l'instruction

int `i`; char `c`;	Valeur initiale	Valeur finale	Pas d'incrémentation	Nombre de boucles	Valeurs prises par `i` ou `c`
for (i = 0; i < 5; i = i +1)	0	4	1	5	0, 1, 2, 3, 4
for (i = 4; i <= 12; i = i + 2)	4	12	2	5	4, 6, 8, 10, 12
for (c = ’a’; c < ’f’; c = c + 1)	’a’	’e’	1	5	a, b, c, d, e
for (i = 5; i > 0 ; i = i – 1)	5	0	– 1	5	5, 4, 3, 2, 1

`i = i + 1` s'écrit plus simplement `i++`, et l'instruction `i--` a le même résultat que l'instruction `i = i - 1`.

Rechercher le code Unicode d'un caractère donné

L'objectif de cet exemple est d'apprendre à construire une boucle `for` et de s'initier à la recherche d'information dans un ensemble de données. Pour cela, nous allons écrire un programme qui recherche dans la table Unicode le code d'un caractère donné par l'utilisateur. Cette recherche s'effectue en comparant chaque caractère de la table Unicode au caractère saisi.

Cahier des charges

La méthode est la suivante :

1. Lire le caractère dont on souhaite connaître le code Unicode.
2. Pour chaque caractère de la table Unicode :

   Si le caractère Unicode est identique au caractère choisi, afficher son code Unicode.

Reprenons chaque point, pour le traduire en un programme Java.

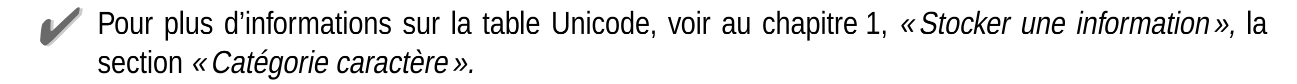
1. Pour lire au clavier le caractère dont on souhaite connaître le code Unicode, les instructions sont les suivantes.

   ✔ Pour plus d'informations, voir le chapitre 2, « *Communiquer une information* ».

```
System.out.println("Quel caractere recherchez-vous : ");
recherche = Lire.c();
```

   Où la variable `recherche` est déclarée de type `char`.

2. Le programme parcourt la table Unicode caractère par caractère et recherche le caractère souhaité. Cette opération est répétitive et s'exécute autant de fois qu'il y a de caractères dans la table Unicode, c'est-à-dire du caractère `0` au caractère `255`.



   ✔ Pour plus d'informations sur la table Unicode, voir au chapitre 1, « *Stocker une information* », la section « *Catégorie caractère* ».

   Pour parcourir cette table, la solution est d'utiliser une boucle `for`, dont la valeur de l'indice varie de 1 en 1, dans l'intervalle [`0`, `255`]. Cette boucle s'écrit :

```
for (i = 1; i < 255; i++)
```

La variable `i`, déclarée de type `int`, représente l'indice du caractère dans la table Unicode. Il y a équivalence entre l'indice et le caractère. En effet, un caractère est défini à partir d'une valeur numérique.

La seule différence entre une valeur numérique et un caractère provient du type de codage utilisé pour les représenter l'un et l'autre. Pour connaître le caractère correspondant à cet indice, la méthode consiste à transformer la valeur de l'indice en un code caractère par l'intermédiaire du cast `(char)`. Ainsi, l'instruction :

```
atrouver = (char) i;
```

transforme l'indice `i` de la table Unicode en son code caractère. La variable `atrouver`, déclarée de type `char`, prend la valeur de ce code.

Connaissant le caractère à rechercher ainsi que le code caractère de chaque caractère de la table Unicode, il suffit de les comparer pour savoir s'ils sont identiques ou non. L'instruction s'écrit sous la forme du test suivant :

```
if (atrouver == recherche)
```

Si le caractère Unicode est identique au caractère choisi, le programme affiche son code Unicode à l'aide des instructions :

```
System.out.print("le code Unicode de " + atrouver);
System.out.println(" est \\u00" + Integer.toString(i,16));
```

Rappelons que le code Unicode d'un caractère s'obtient en plaçant derrière les caractères `\u00`, la valeur hexadécimale de la position du caractère dans la table Unicode. Pour afficher ce code, nous devons donc traduire la variable `i` (qui correspond à la position du caractère dans la table Unicode) en valeur hexadécimale. Cette traduction est réalisée par la fonction :

```
Integer.toString(valeur entière, base)
```

qui transforme le paramètre `valeur entière` en une chaîne de caractères suivant le codage donné par le paramètre `base`. Si `valeur entière` représente l'indice `i` et que `base` prenne la valeur 16, nous obtenons la valeur hexadécimale de la position du caractère trouvé.

La suite des caractères `\u00` placée dans la fonction `System.out.println` est considérée comme une séquence particulière puisqu'elle permet l'affichage des caractère spéciaux. Pour annuler le caractère spécifique de cette séquence, il est nécessaire de placer un premier `\` devant `\u00`.

Exemple : code source complet

Pour obtenir un programme à part entière, l'ensemble des instructions développées au cours de la section précédente est à placer dans une fonction `main()` et une classe, comme ci-dessous :

```
public class QuelUnicode
{
 public static void main (String [] parametre)
 {
```

```
   int i;
   char recherche, atrouver;
   System.out.println("Quel caractere recherchez-vous : ");
   recherche = Lire.c();
   for (i = 1; i < 255; i++)
   {
    atrouver = (char) i;
    if (atrouver == recherche)
    {
     System.out.print("le code Unicode de " + atrouver);
     System.out.println(" est \\u00" + Integer.toString(i,16));
    } // Fin du if
   } // Fin du for
  } // Fin du main()
}// Fin de QuelUnicode
```

Résultat de l’exécution

Les valeurs grisées correspondent aux valeurs saisies par l’utilisateur. Suivant l’environnement d’exécution, le programme donne des résultats différents.

Exécution sous Dos

```
Quel caractere recherchez-vous : é
le code Unicode de é est \u0082
```

Exécution sous Mac OS

```
Quel caractere recherchez-vous : é
le code Unicode de é est \u00c8
```

Exécution sous Windows ou Unix

```
Quel caractere recherchez-vous : é
le code Unicode de é est \u00e9
```

L’exécution du même programme sur les différents environnements montre bien que le code Unicode d’un caractère spécial (par exemple accentué) n’est pas le même d’un environnement à un autre.

Quelle boucle choisir ?

Chacune des trois boucles étudiées dans ce chapitre permet de répéter un ensemble d’instructions. Cependant, les différentes propriétés de chacune d’entre elles font que le programmeur utilisera un type de boucle plutôt qu’un autre, suivant le problème à résoudre.

Choisir entre une boucle `do while` et une boucle `while`

Les boucles `do while` et `while` se ressemblent beaucoup dans leur syntaxe, et il paraît parfois difficile au programmeur débutant de choisir l’une plutôt que l’autre.

Remarquons cependant que la différence essentielle entre ces deux boucles réside dans la position du test de sortie de boucle. Pour la boucle `do while`, la sortie de boucle s'effectue en fin de boucle, alors que, pour la boucle `while`, la sortie de boucle se situe dès l'entrée de la boucle.

De ce fait, la boucle `do while` est plus souple à manipuler, les instructions qui la composent étant exécutées au moins une fois, quoi qu'il arrive. Pour la boucle `while`, il est nécessaire de veiller à l'initialisation de la variable figurant dans le test d'entrée de boucle, de façon à être sûr d'exécuter au moins une fois les instructions composant la boucle.

Certains algorithmes demandent à ne jamais répéter, sous certaines conditions, un ensemble d'instructions. Dans de tels cas, la structure `while` est préférable à la structure `do while`.

Choisir entre la boucle `for` et `while`

Les boucles `for` et `while` sont équivalentes. En effet, en examinant les deux boucles du tableau ci-dessous,

La boucle `for`	La boucle `while`

`int i;`
`for (i = 0; i <= 10; i = i+1 )`
`{`
`}`
	`int i = 0`
	`while (i <= 10)`
	`{`
	`i = i+1;`
	`}`

nous constatons que, pour chacune d'entre elles, la boucle débute avec `i = 0`, puis, tant que `i` est inférieur ou égal à `10`, `i` est incrémenté de `1`.

Malgré cette équivalence, pour choisir entre une boucle `for` et une boucle `while`, remarquons que :

- La boucle `for` est utilisée quand on connaît à l'avance le nombre d'itérations à exécuter.
- La boucle `while` est employée lorsque le nombre d'itérations est laissé au choix de l'utilisateur du programme ou déterminé à partir du résultat d'un calcul réalisé au cours de la répétition.

Résumé

En langage Java, il existe trois types de structures pour réaliser des répétitions. Elles sont décrites par les instructions : `do...while`, `while` et `for`.

- La boucle `do...while` (faire... tant que) permet d'exécuter les instructions situées dans le bloc défini par des `{}`, tant que l'expression conditionnelle placée entre `()` est vraie.

```
do {
    plusieurs instructions;
} while (expression);
```

Les instructions sont exécutées au moins une fois puisque l'expression conditionnelle est examinée en fin de boucle, après exécution des instructions.

- La boucle `while` (tant que) permet d'exécuter les instructions situées dans le bloc défini par `{}`, tant que l'expression conditionnelle placée entre `()` est vraie.

```
while (expression)
{
    plusieurs instructions;
}
```

L'expression conditionnelle étant examinée en début de boucle, les instructions situées dans le bloc peuvent ne pas être exécutées si la condition n'est pas vérifiée dès le début.

- La boucle `for` permet d'écrire des boucles dont on connaît à l'avance le nombre d'itérations à exécuter. Elle est équivalente à l'instruction `while` mais est plus simple à écrire.

```
for (initialisation; condition; incrément)
{
    plusieurs instructions;
}
```

Les termes `Initialisation`, `Condition` et `Incrément` sont des instructions séparées obligatoirement par des points-virgules (`;`). Ces instructions définissent un indice qui contrôle le bon déroulement de la boucle. Ainsi :

  – `Initialisation` permet d'initialiser la variable représentant l'indice de la boucle.
  – `Condition` définit la condition à vérifier pour continuer à exécuter la boucle.
  – `Incrément` permet d'augmenter ou de diminuer de *N* la valeur de la variable représentant l'indice de la boucle. *N* est appelé le pas d'incrémentation.

À partir des structures répétitives nous avons également abordé la notion de comptage de valeurs, c'est-à-dire :

- Le **comptage** d'un certain nombre de valeurs (par exemple, compter le nombre de notes d'un étudiant). Pour cela, il suffit d'employer une variable entière initialisée à `0` avant d'entamer la boucle. La variable augmente de `1` à l'intérieur de la boucle à l'aide de l'instruction `i = i + 1` (en supposant que `i` soit notre variable compteur). On dit alors que la variable `i` est incrémentée de `1`.
- L'**accumulation** de valeurs (par exemple, faire la somme des notes d'un étudiant). Cette technique est réalisée à l'aide d'une variable entière initialisée à `0` avant d'entamer la boucle. La variable augmente de la valeur de la variable à accumuler (de la valeur de la note, par exemple), à l'intérieur de la boucle. Cette augmentation s'effectue à l'aide de l'instruction `s = s + valeur`, en supposant que `s` soit notre variable d'accumulation et `valeur` la variable représentant la valeur à accumuler.

Remarquons, pour finir, que l'instruction `i++` est l'équivalent simplifié de `i = i + 1`, tandis que `i--` est l'équivalent simplifié de `i = i - 1`.

Exercices

Comprendre la boucle `do…while`

4.1 Afin d'exécuter le programme suivant :

```
public class Exercice1
{
 public static void main (String [] argument)
 {
  int a,b,r;
  System.out.println("Entrer un entier : ");
  a = Lire.i();
  System.out.println("Entrer un entier : ");
  b = Lire.i();
  do
  {
    r = a%b;
    a = b;
    b = r;
   } while (r !=0 );
  System.out.println("Le resultat est " + a);
 }
}
```

a. Examinez le code source (programme), repérez les instructions concernées par la boucle répétitive, et déterminez les instructions de début et fin de boucle.

b. Quelle est l'instruction qui permet de modifier le résultat du test de sortie de boucle ?

c. En supposant que l'utilisateur entre les valeurs `30` et `42`, exécutez le programme à la main (pour vous aider, construisez le tableau d'évolution de chaque variable déclarée).

d. En supposant que l'utilisateur entre les valeurs `35` et `6`, exécutez le programme à la main (pour vous aider, construisez le tableau d'évolution de chaque variable déclarée).

e. Quel est le calcul réalisé par ce programme ?

Apprendre à compter, accumuler et rechercher une valeur

4.2 Écrivez en français, en faisant ressortir la structure répétitive de la marche à suivre, le programme résolvant les quatre points suivants :

a. Lire un nombre quelconque de valeurs entières non nulles. La saisie des valeurs se termine lorsqu'on entre la valeur `0`.

b. Afficher la plus grande des valeurs.

c. Afficher la plus petite des valeurs.

d. Calculer et afficher la moyenne de toutes les valeurs.

4.3 Traduisez la marche à suivre précédente en un programme Java. Utilisez pour cela une boucle `do…while`. Pour trouver la plus grande ou la plus petite valeur, vous pouvez vous aider de l'exemple « Rechercher le plus grand de deux éléments », décrit au cours du chapitre 3, « Faire des choix ».

Comprendre la boucle `while`*, traduire une marche à suivre en programme Java*

4.4 Écrivez un programme `Devinette`, qui tire un nombre au hasard entre 0 et 10 et demande à l'utilisateur de trouver ce nombre. Pour ce faire, la méthode est la suivante :

a. Tirer au hasard un nombre entre 0 et 10.

b. Lire un nombre.

c. Tant que le nombre lu est différent du nombre tiré au hasard :

– Lire un nombre.

– Compter le nombre de boucle.

d. Afficher un message de réussite ainsi que le nombre de boucles.

4.5 Reprenez chaque point énoncé ci-dessus, et traduisez-le en langage Java. Notez que, pour tirer un nombre au hasard entre 0 et 10, l'instruction s'écrit :

```
i = (int) (10*Math.random());
```

où `i` est une variable entière qui reçoit la valeur tirée au hasard.

4.6 Déclarez toutes les variables utilisées dans votre programme en veillant à ce qu'elles soient bien initialisées.

4.7 Lorsque le programme `Devinette` fonctionne bien, modifiez-le de façon que :

a. Les valeurs tirées au hasard soient comprises entre 0 et 50.

b. Un message d'erreur s'affiche si la réponse est mauvaise.

c. Le programme indique si la valeur saisie au clavier est plus grande ou plus petite que la valeur tirée au hasard.

d. À titre de réflexion : comment faut-il s'y prendre pour trouver la valeur en donnant le moins de réponses possibles ?

Comprendre la boucle `for`

4.8 Afin d'exécuter le programme suivant :

```
public class Exercice8
{
 public static void main (String [] parametre)
 {
  long i, b = 1;
  int a;
  System.out.println("Entrer un entier :");
  a = Lire.i();
```

```
    for (i = 2; i <= a; i++)
      b = b * i;
    System.out.println("Le resultat vaut " + b);
  }
}
```

a. Examinez le programme, repérez les instructions concernées par la boucle répétitive, et déterminez les instructions de début et fin de boucle.

b. Quelle est la valeur initiale de `i` et quelle est sa valeur en sortie de boucle ? Combien de boucles sont réalisées ?

c. Quelle est l'instruction qui permet de modifier le résultat du test de sortie de boucle ?

d. En supposant que l'utilisateur entre la valeur `6`, exécutez le programme à la main (pour vous aider, construisez le tableau d'évolution de chaque variable déclarée).

e. Quel est le calcul réalisé par ce programme ?

4.9 En utilisant une boucle `for`, écrivez un programme qui affiche l'alphabet, d'abord à l'endroit, puis à l'envers, après un passage à la ligne.

Le projet « Gestion d'un compte bancaire »

Rendre le menu interactif

Une fois l'affichage du menu réalisé à partir de l'énoncé donné à la fin du chapitre 3, « Faire des choix », le programme exécuté donne à choisir parmi les cinq options suivantes :

```
1. Création d'un compte
2. Affichage d'un compte
3. Créer une ligne comptable
4. Sortir
5. De l'aide

Votre choix :
```

Si l'utilisateur choisit l'`option 1`, le programme lui demande de saisir les données nécessaires à la création du compte (type, numéro, valeur initiale, etc.). Une fois les données saisies, le programme s'arrête. Il n'est pas possible de choisir, par exemple, l'`option 2` pour afficher les valeurs saisies à l'étape précédente.

Pour remédier à cette situation, il est nécessaire de placer les instructions concernées à l'intérieur d'une boucle, de façon à voir réapparaître le menu une fois l'option réalisée. Pour cela, vous devez :

a. Écrire en français la structure répétitive, afin de déterminer la condition de sortie de boucle.

b. Choisir la structure répétitive parmi les trois proposées par le langage Java.

c. Traduire la marche à suivre en programme Java, en prenant soin d'initialiser la variable de contrôle de la boucle et en insérant, à l'intérieur de la boucle, toutes les instructions nécessaires à l'affichage du menu.

PARTIE 2

Initiation à la programmation orientée objet

5

De l’algorithme paramétré à l’écriture de fonctions

L’étude des chapitres précédents montre qu’un programme informatique est constitué d’instructions élémentaires (affectation, comparaison ou encore répétition) et de sous-programmes (calcul de la racine carrée, affichage de données), appelés fonctions ou encore méthodes.

Ces instructions sont de nature suffisamment générale pour s’adapter à n’importe quel problème. En les utilisant à bon escient, il est possible d’écrire des programmes informatiques simples mais d’une grande utilité.

Dans le cadre du développement de logiciels de grande envergure, les programmeurs souhaitent aussi définir leurs propres instructions, adaptées au problème qu’ils traitent. Pour cela, les langages de programmation offrent la possibilité de créer des fonctions spécifiques, différentes des fonctions prédéfinies par le langage.

Pour comprendre l’intérêt des fonctions, nous analysons d’abord le concept d’algorithme paramétré à partir d’un exemple imagé.

Ensuite, nous étudions la bibliothèque de fonctions mathématiques définie dans le langage Java (*section « Des fonctions Java prédéfinies »*). Cette étude montre les principes d’utilisation de ces fonctions et explique comment élaborer et construire vos fonctions (*section « Construire ses propres fonctions »*).

Pour finir, nous examinons comment la construction et l’utilisation de fonctions font évoluer la structure générale d’un programme (*section « Les fonctions au sein d’un programme Java »*).

Algorithme paramétré

Certains algorithmes peuvent être appliqués à des problèmes voisins en modifiant simplement les données pour lesquels ils ont été construits. En faisant varier certaines valeurs, le programme fournit un résultat différent du précédent. Ces valeurs, caractéristiques du problème à traiter, sont appelées paramètres du programme.

Pour comprendre concrètement ce concept, nous allons reprendre l'algorithme du café chaud pour le transformer en un algorithme qui nous permettra de faire du thé ou du café chaud.

Faire un thé chaud, ou comment remplacer le café par du thé

Faire un café chaud ou faire un thé chaud est une opération à peu près semblable. En reprenant la liste de toutes les opérations nécessaires à la réalisation d'un café chaud, nous remarquons qu'en remplaçant simplement le mot café par le mot thé, nous obtenons du thé chaud.

Instructions	Bloc d'instructions
0. Prendre une cafetière. 1. Poser la cafetière sur la table. 2. Prendre du **thé**. 3. Prendre un filtre. 4. Verser le **thé** dans le filtre. 5. Prendre de l'eau. 6. Verser l'eau dans la cafetière. 7. Brancher la cafetière. 8. Allumer la cafetière. 9. Attendre que le **thé** soit prêt 10. Prendre une tasse. 11. Poser la tasse sur la table. 12. éteindre la cafetière. 13. Verser le **thé** dans la tasse.	Préparer le thé

Cette recette n'est certes pas traditionnelle, mais elle a le mérite d'être pédagogiquement simple. Pour faire du café ou du thé, il suffit d'employer la même recette, ou méthode, en prenant comme ingrédient du café ou du thé, selon notre choix.

Dans le monde réel, le fait de remplacer un ingrédient par un autre ne pose pas de difficultés particulières. Dans le monde informatique, c'est plus complexe. En effet, l'ordinateur ne fait qu'exécuter la marche à suivre fournie par le programmeur. Dans notre cas, pour avoir du café ou du thé, le programmeur doit écrire la marche à suivre pour chacune des boissons. La tâche est fastidieuse, puisque chacun des programmes se ressemble, tout en étant différent sur un détail (café ou thé).

Définir les paramètres

Pour éviter d'avoir à recopier chaque fois des marches à suivre qui ne diffèrent que sur un détail, l'idée est de construire un algorithme général. Cet algorithme ne varie

qu'en fonction d'ingrédients déterminés, qui font que le programme donne un résultat différent.

En généralisant l'algorithme du thé ou du café chaud, on exprime une marche à suivre permettant de réaliser une boisson chaude. Pour obtenir un résultat différent (café ou thé), il suffit de définir comme paramètre de l'algorithme l'ingrédient, café ou thé, à choisir.

La marche à suivre s'écrit en remplaçant les mots café ou thé par le mot **ingrédient**.

Instructions	Nom du bloc d'instructions
0. Prendre une cafetière. 1. Poser la cafetière sur la table. 2. Prendre **ingrédient**. 3. Prendre un filtre. 4. Verser **ingrédient** dans le filtre. 5. Prendre de l'eau. 6. Verser l'eau dans la cafetière. 7. Brancher la cafetière. 8. Allumer la cafetière. 9. Attendre que **ingrédient** soit prêt. 10. Prendre une tasse. 11. Poser la tasse sur la table. 12. éteindre la cafetière. 13. Verser **ingrédient** dans la tasse.	Préparer **(ingrédient)**

Faire du café équivaut donc à exécuter le bloc d'instructions `Préparer (ingrédient)` en utilisant comme ingrédient du café. L'exécution du bloc `Préparer (le café)` a pour conséquence de réaliser les instructions `2`, `4`, `9` et `13` du bloc d'instructions avec comme ingrédient du café. L'instruction `2,` par exemple, s'exécute en remplaçant le terme `ingrédient` par `le café`. Au lieu de lire `prendre ingrédient`, il faut lire `prendre le café`.

De la même façon, faire du thé revient à exécuter le bloc d'instructions `Préparer (le thé)`. Le paramètre `ingrédient` correspond ici au thé, et les instructions `2`, `4`, `9` et `13` sont exécutées en conséquence.

Suivant la valeur prise par le paramètre `ingrédient`, l'exécution de cet algorithme fournit un résultat différent. Ce peut être du café ou du thé.

Donner un nom au bloc d'instructions

Nous constatons qu'en paramétrant un algorithme, nous n'avons plus besoin de recopier plusieurs fois les instructions qui le composent pour obtenir un résultat différent.

En donnant un nom au bloc d'instructions correspondant à l'algorithme général Préparer(), nous définissons un sous-programme capable d'être exécuté autant de fois que nécessaire. Il suffit pour cela d'appeler le sous-programme par son nom.

De plus, grâce au paramètre placé entre les parenthèses qui suivent le nom du sous-programme, la fonction s'exécute avec des valeurs différentes, modifiant de ce fait le résultat.

Un algorithme paramétré est défini par :

- un nom ;
- un ou plusieurs paramètres.

En fin d'exécution, il fournit :

- un résultat, qui diffère suivant la valeur du ou des paramètres.

Dans le langage Java, les algorithmes paramétrés s'appellent des **fonctions** ou encore des **méthodes**. Grâce à elles, il est possible de traduire un algorithme paramétré en programme informatique. Avant d'examiner comment écrire ces algorithmes en langage Java, nous allons tout d'abord étudier les fonctions prédéfinies du langage Java, de façon à mieux comprendre comment elles s'utilisent.

Des fonctions Java prédéfinies

Un grand nombre de programmes informatiques font appel à des calculs mathématiques simples, tels que le calcul d'un sinus ou d'une racine carrée. Pour trouver la valeur d'un sinus, par exemple, le programmeur n'a pas, fort heureusement, à réécrire pour chaque programme l'algorithme mathématique du calcul d'un sinus. Les fonctions mathématiques sont déjà programmées.

Le langage Java propose un ensemble de fonctions prédéfinies, mathématiques ou autres, très utiles, comme nous le verrons au cours des chapitres suivants. Notre objectif n'est pas de décrire l'intégralité des fonctions disponibles, car ce seul manuel n'y suffirait pas. Nous souhaitons faire comprendre la manipulation de ces fonctions. Pour ce faire, nous allons étudier une partie de la librairie mathématique de Java, appelée `Math`, et déterminer ensuite les principes généraux d'utilisation des fonctions.

La librairie `Math`

La librairie mathématique du langage Java est composée d'un ensemble de fonctions prédéfinies, qui permettent de calculer toutes sortes d'équations mathématiques. Parmi ces fonctions, se trouvent les fonctions trigonométriques (sinus, cosinus, tangente, etc.), logarithmiques, d'arrondis, de calcul de puissances ou de racines carrées.

Ces fonctions sont regroupées dans la bibliothèque de programmes `Math`. Le nom de chaque fonction débute toujours par le terme `Math`, suivi d'un point puis du nom de la fonction. Ce nom commence toujours par une minuscule. Voici une liste partielle des fonctions qui composent la librairie `Math` :

Fonctions trigonométriques

Opération mathématique	Fonction Java
Calculer le cosinus d'un angle (radian)	`Math.cos()`
Calculer le sinus d'un angle (radian)	`Math.sin()`
Calculer la tangente d'un angle (radian)	`Math.tan()`

Fonctions logarithmiques

Opération mathématique	Fonction Java
Calculer le logarithme d'une valeur	`Math.log()`
Calculer l'exponentielle d'un nombre	`Math.exp()`

Calcul d'arrondis

Opération	Fonction Java
Arrondir à l'entier inférieur	`Math.floor()`
Arrondir à l'entier supérieur	`Math.ceil()`

Autres calculs mathématiques

Opération mathématique	Fonction Java
Calculer la racine carrée d'un nombre	`Math.sqrt()`
a^b (a puissance b)	`Math.pow()`
|a| (valeur absolue de a)	`Math.abs()`

Divers

Opération	Fonction Java
Trouver la plus grande de deux valeurs	`Math.max()`
Trouver la plus petite de deux valeurs	`Math.min()`
Tirer un nombre au hasard entre 0 et 1	`Math.random()`

Exemples d'utilisation

Ces fonctions s'utilisent en plaçant dans le programme Java le nom d'appel de la fonction. Voici en exemple un programme qui utilise l'ensemble des fonctions décrites ci-dessus :

Exemple : code source complet

```
public class FonctionMatématique
{
 public static void main(String [] argument)
 {
  double résulat, a, b;
  System.out.print("Entrez une premiere valeur :");
  a = Lire.d();
  System.out.print("Entrez une seconde valeur :");
  b = Lire.d();
```

```
    résultat = Math.cos(a) ;
    System.out.println("Cos(" + a + ") = " + résultat);
    résultat = Math.sin(a) ;
    System.out.println("Sin(" + a + ") = " + résultat);
    résultat = Math.tan(a) ;
    System.out.println("Tan(" + a + ") = " + résultat);
    résultat = Math.log(a) ;
    System.out.println("Log(" + a + ") = " + résultat);
    résultat = Math.exp(a) ;
    System.out.println("Exp(" + a + ") = " + résultat);
    résultat = Math.floor(a) ;
    System.out.println("Floor(" + a + ") = " + résultat);
    résultat = Math.ceil(a) ;
    System.out.println("Ceil(" + a + ") = " + résultat);
    résultat = Math.sqrt(a) ;
    System.out.println("Sqrt(" + a + ") = " + résultat);
    résultat = Math.pow(a,b) ;
    System.out.println("Pow(" + a + ", " + b +") = " + résultat);
    résultat = Math.abs(a) ;
    System.out.println("Abs(" + a + ") = " + résultat);
    résultat = Math.max(a,b) ;
    System.out.println("Max(" + a + ", " + b + ") = " + résultat);
    résultat = Math.min(a,b) ;
    System.out.println("Min(" + a + ", " + b + ") = " + résultat);
    résultat = Math.random() ;
    System.out.println("Random() = " + résultat);
  }
}
```

Une fois les instructions de ce programme compilées, l'interpréteur Java les exécute une à une. Le résultat est le suivant :

Résultat de l'exécution

Les caractères grisés sont des valeurs choisies par l'utilisateur.

```
Entrez une premiere valeur : 0.1
Entrez une seconde valeur : 2
Cos(0.1) = 0.9950041652780257
Sin(0.1) = 0.09983341664682815
Tan(0.1) = 0.10033467208545055
Log(0.1) = -2.3025850929940455
Exp(0.1) = 1.1051709180756477
Floor(0.1) = 0.0
Ceil(0.1) = 1.0
Sqrt(0.1) = 0.31622776601683794
Pow(0.1, 2.0) = 0.01
Abs(0.1) = 0.1
Max(0.1, 2.0) = 2.0
Min(0.1, 2.0)= 0.1
Random() = 0.6993848420032578
```

Principes de fonctionnement

L'étude de ce programme met en évidence plusieurs aspects importants concernant l'utilisation des fonctions et leur mode de fonctionnement.

Le nom des fonctions

- Le nom de chaque fonction est défini par le langage Java. Pour connaître le nom des différentes fonctions proposées par le Java, il est nécessaire de consulter l'aide en ligne du compilateur ou le site Internet de Sun (*voir le CD-Rom livré avec cet ouvrage*), ou encore des livres plus spécifiques sur le langage Java et les bases de données ou les réseaux.
- Remarquons que l'exécution d'une fonction passe par l'écriture dans une instruction du nom de la fonction choisie, suivi de paramètres éventuels placés entre parenthèses.

Mémoriser le résultat d'une fonction

- Pour mémoriser le résultat du calcul, la fonction est placée dans une instruction d'affectation. La fonction, située à droite du signe `=`, est exécutée en premier. Après quoi, la variable située à gauche du signe `=` récupère la valeur calculée lors de l'exécution de la fonction.

> ✔ Pour plus d'informations voir au chapitre 1, *« Stocker une information »*, la section *« Rôle et mécanisme de l'affectation »*.

- Dans notre exemple, toutes les fonctions de la bibliothèque `Math` fournissent en résultat une valeur numérique de type `double`. En conséquence, la variable `résultat`, qui récupère le résultat de chaque fonction, est déclarée de type `double`.

Les paramètres d'une fonction

Les fonctions possèdent zéro, un, voire deux paramètres. Ainsi :

- La fonction `Math.random()` ne possède pas de paramètre. Cette fonction donne en résultat une valeur au hasard, comprise entre 0.0 et 1.0, indépendamment de toute condition. Aucun paramètre n'est donc nécessaire à sa bonne marche.

  Remarquons que même si la fonction n'a pas de paramètre, il reste nécessaire de placer des parenthèses, ouvrante puis fermante, derrière le nom d'appel de la fonction. Si aucune parenthèse n'est placée, le compilateur ne considère pas le terme `Math.random` comme une fonction mais comme un nom de variable.

  Toute fonction possède dans son nom d'appel des parenthèses, ouvrante puis fermante.

- La fonction `Math.sqrt()`ne comporte qu'un seul paramètre, puisqu'elle calcule la racine carrée d'un seul nombre à la fois. Il est possible de placer entre parenthèses une expression mathématique plutôt qu'un paramètre. Ainsi, l'expression `Math.sqrt (b*b - 4*a*c)` permet le calcul de la racine carrée du discriminant d'une équation du second degré.

  Observons que le paramètre placé entre parenthèses dans la fonction `Math.sqrt()` est de type `double`. De cette façon, il est possible de calculer la racine carrée de tout type

de valeur numérique, les types `byte`, `short`, `int` ou `long` se transformant sans difficulté en type `double`.


Pour plus d'informations, voir, au chapitre 1, *« Stocker une information »,* la section *« La transformation de types ».*

Il n'est pas permis de placer en paramètre un caractère, une suite de caractères ou un booléen. Par exemple, le fait d'écrire `Math.sqrt("Quatre")` entraîne une erreur en cours de compilation, l'ordinateur ne sachant pas transformer le mot « Quatre » en la valeur numérique 4 (message d'erreur : `Incompatible type for method. Can't convert java.lang.String to double`) .

Dans l'appel de la fonction, le type des paramètres doit être respecté, sous peine d'obtenir une erreur de compilation.

- La fonction `Math.pow(a,b)` possède deux paramètres pour calculer a^b (`a` à la puissance `b`). Ces paramètres sont séparés par une virgule. Si les valeurs `a` et `b` sont inversées dans l'appel de la fonction (`Math.pow(b,a)`), le calcul effectué a pour résultat b^a (`b` à la puissance `a`).

  Dans l'appel de la fonction, l'ordre des paramètres doit être respecté, sous peine d'obtenir un résultat différent de celui attendu.

Les fonctions étudiées dans cette section sont des fonctions prédéfinies par le langage Java. Le programmeur les utilise en connaissant le résultat qu'il souhaite obtenir. Les programmes ainsi écrits sont constitués d'instructions simples et d'appels à des fonctions connues du langage Java.

Le langage Java offre aussi au programmeur la possibilité d'écrire ses propres fonctions de façon à obtenir différents programmes adaptés au problème qu'il doit résoudre. Nous étudions cette technique à la section qui suit.

Construire ses propres fonctions

Une fonction développée par un programmeur s'utilise de la même façon qu'une fonction prédéfinie. Elle s'exécute en plaçant l'instruction d'appel à la fonction dans le programme. Cette étape est décrite à la section « Appeler une fonction ».

Pour que l'ordinateur puisse lire et exécuter les instructions composant la fonction, il convient de définir cette dernière, c'est-à-dire d'écrire une à une les instructions qui la composent. Plusieurs étapes sont nécessaires à cette définition. Nous les étudions à la section « Définir une fonction ».

Pour mieux cerner les difficultés liées à ces opérations, nous allons prendre comme exemple la création d'une fonction qui calcule le périmètre d'un cercle de rayon quelconque.

Appeler une fonction

Toute fonction possède un nom d'appel, qui permet de l'identifier. Ce nom est choisi de façon à représenter et résumer tout ce qui est réalisé par son intermédiaire. Dans notre exemple, nous devons calculer le périmètre d'un cercle. Nous appelons donc la fonction qui réalise ce calcul, c'est-à-dire `périmètre()`.

D'une manière générale, une fonction représente une action. C'est pourquoi le choix d'un verbe comme nom de fonction permet de mieux symboliser les opérations réalisées. Ici, le terme `périmètre()` n'est pas un verbe, mais il faut comprendre par `périmètre()` l'action de calculer le périmètre.

Le nom de la fonction `périmètre()` étant défini, nous souhaitons calculer le périmètre d'un cercle dont la valeur du rayon soit saisie au clavier. Pour cela, observons le programme qui calcule la racine carrée d'un nombre saisi au clavier :

```
double résulat, a;
System.out.print("Entrez une valeur :");
a = Lire.d();
résultat = Math.sqrt(a) ;
System.out.println("Sqrt(" + a + ") = " + résultat);
```

L'instruction `résultat = Math.sqrt(a) ;` calcule la racine carrée du nombre `a`, dont la valeur a été saisie au clavier à l'instruction précédente. Elle place ensuite le résultat de ce calcul dans la variable `résultat`.

En modifiant le nom d'appel de la fonction `Math.sqrt()` par `périmètre()`, nous obtenons un programme qui appelle la fonction `périmètre()` et qui, par conséquent, calcule le périmètre d'un cercle dont la valeur du rayon `a`, est saisie au clavier. La valeur du périmètre est placée dans la variable `résultat` par l'intermédiaire du signe d'affectation `=`.

Pour notre exemple, le programme d'appel à la fonction `périmètre()` s'écrit :

```
public static void main(String [] parametre)
{
 // Déclaration des variables
 double résultat ;
 int valeur ;
 System.out.print("Valeur du rayon : ");
 valeur = Lire.i();
 résultat = périmètre (valeur);
 System.out.print("rayon = " + valeur + " perimetre = " + résultat);
}
```

Le programme ainsi écrit permet de calculer le périmètre d'un cercle de rayon donné, à la seule condition de définir la fonction `périmètre()`dans le programme. En effet, cette fonction n'est pas prédéfinie dans le langage Java, et il est nécessaire de détailler les instructions qui la composent.

Sans cette définition, l'ordinateur n'est pas à même de déterminer par lui-même les instructions à exécuter, et le message d'erreur `Method perimetre(int) not found in class Cercle` apparaît en cours de compilation.

Définir une fonction

La définition d'une fonction fournit à l'ordinateur les instructions à exécuter lors de l'appel de la fonction. Cette opération passe par les étapes suivantes :

- déterminer les instructions composant la fonction ;
- associer le nom de la fonction aux instructions ;

- établir les paramètres utiles à l'exécution de la fonction ;
- préciser le type de résultat fourni par la fonction.

De façon à mieux comprendre le rôle de chacune de ces étapes, définissons la fonction qui calcule le périmètre d'un cercle de rayon quelconque.

Déterminer les instructions composant la fonction

Pour sélectionner les instructions utiles au calcul du périmètre d'un cercle, reprenons le programme `Cercle`.

✔ Voir au chapitre introductif, *« Naissance d'un programme »*, la section *« Un premier programme en Java »*.

```
public class Cercle
{
  public static void main(String [] argument)
  {
   // Déclaration des variables
   double r, p ;
   // Afficher le message "Valeur du rayon : " à l'écran
   System.out.print("Valeur du rayon : ") ;
   // Lire au clavier une valeur, placer cette valeur dans la variable R
   r= Lire.d() ;
   // Calculer la circonférence en utilisant la formule consacrée
   p = 2*Math.PI*r ;
   // Afficher le résultat
   System.out.print("Le cercle de rayon "+ R +" a pour perimetre : "+ P);
  }
}
```

Nous avons observé, lors de la mise en œuvre d'algorithmes paramétrés, que la marche à suivre décrivant l'algorithme devait être la plus générale possible (*voir la section « Définir les paramètres »*). C'est pourquoi, pour notre cas, seules les instructions :

```
// Déclaration des variables
   double r, p ;
// Calculer la circonférence en utilisant la formule consacrée
   p = 2*Math.PI*r ;
```

sont utilisées dans la fonction de calcul du périmètre d'un cercle. Les instructions relatives à la demande de saisie d'une valeur au clavier ne sont pas à placer dans la fonction. Pour vous en convaincre, observez que l'ordinateur, à l'appel de la fonction `Math.sqrt()`, ne demande pas de valeur à saisir. Il ne fait que calculer la racine carrée d'une valeur passée en paramètre.

Les instructions ainsi choisies sont placées dans ce que l'on appelle, dans le jargon informatique, le **corps de la fonction**, et ce de la façon suivante :

```
// Définition du corps de la fonction
{ // début de la fonction
  double p, r;
  p = 2 * Math.PI * r;
} // fin de la fonction
```

Le corps de la fonction est déterminé par les accolades `{ et }`. Les instructions qui le composent sont ici des déclarations de variables et des instructions d'affectation. Dans d'autres cas, peuvent aussi figurer des instructions de test, de répétition, etc.

Associer le nom aux instructions

Une fois écrit le corps de la fonction, il est nécessaire de l'associer au nom d'appel de la fonction.

Le nom d'une fonction est lié au bloc d'instructions qui la compose, grâce à un **en-tête de fonction**. Ce dernier a pour forme :

```
public static type nomdelafonction (paramètres)
```

L'en-tête d'une fonction permet de préciser :

- Le nom de la fonction (pour notre exemple le `nomdelafonction` est `périmètre`).
- Les `paramètres` éventuels de la fonction.
- Le `type` de résultat fourni par la fonction.

Les mots-clés `public static` sont à placer pour l'instant obligatoirement devant le type de résultat de la fonction.



Nous expliquons la présence de ces termes à la section *« Collectionner un nombre fixe d'objets »* du chapitre 9, *« La ligne de commande »,* car ils sont liés aux concepts de la programmation objet.

L'en-tête d'une fonction se place, comme son nom l'indique, au-dessus du corps de la fonction. Pour notre exemple, il se place de la façon suivante :

```
// En-tête de la fonction
public static type périmètre (paramètres)
{ // début de la fonction
  double p, r;
  p = 2 * Math.PI * r;
} // fin de la fonction
```

De cette façon, le corps de la fonction est associé au nom `périmètre()`. À l'appel du nom de la fonction `périmètre()`, l'ordinateur exécute les instructions placées dans le corps de la fonction.

Établir les paramètres utiles

Comme nous venons de le voir, le périmètre du cercle est calculé à partir du rayon, dont la valeur est saisie avant l'appel de la fonction. La valeur du rayon est placée en paramètre de la fonction, comme lors du calcul de la racine carrée d'un nombre.

Le rayon du cercle est considéré comme le paramètre de la fonction `périmètre()`, et l'en-tête de la fonction s'écrit comme suit :

```
public static type périmètre (int r)
```

Comme la variable `r` est déclarée à l'intérieur des parenthèses de la fonction `périmètre()`, elle est considérée par le compilateur Java comme étant le paramètre de la fonction `périmètre()`.

L'instruction de déclaration, située dans le corps de la fonction, doit être ainsi modifiée :

```
double p;
```

La variable `r` est déclarée dans l'en-tête de la fonction, et elle ne peut donc être déclarée une deuxième fois à l'intérieur de la fonction, sous peine de provoquer une erreur de compilation (message d'erreur : `variable 'r' is already defined in this method`).

Le paramètre `r` est aussi appelé **paramètre formel**. Il prend la forme (la valeur) de la variable donnée au moment de l'appel de la fonction. Pour bien comprendre cela, rappelons-nous de l'algorithme du café ou du thé chaud, dans lequel nous avons utilisé une variable `ingrédient` prenant la forme de café ou de thé suivant ce que l'on souhaitait obtenir. Ici, `r` prend la valeur de la variable `valeur` lors de l'appel de la fonction `résultat = périmètre(valeur)` depuis la fonction `main()`.

Remarquons aussi que le paramètre `valeur` fourni lors de l'appel de la fonction `périmètre ()` est appelé **paramètre réel** ou encore **paramètre effectif**. C'est la valeur de ce paramètre qui est transmise au paramètre formel lors de l'appel de la fonction.

Préciser le type de résultat fourni

Une fois le périmètre calculé grâce à l'instruction :

```
p = 2 * Math.PI * r;
```

la valeur contenue dans la variable `p` doit être transmise et placée dans la variable `résultat`, déclarée dans le programme décrit à la section « Appeler une fonction » de ce chapitre. Pour ce faire, les deux opérations suivantes sont à réaliser :

- Placer une instruction `return`, suivie de la variable contenant le résultat en fin de fonction. Pour notre cas :

  ```
  return p;
  ```

  À la lecture de cette instruction, le programme sort de la fonction `périmètre()` et transmet la valeur contenue dans la variable `p` au programme qui a appelé la fonction `périmètre()`.

- Spécifier le type de la valeur retournée dans l'en-tête de la fonction. Pour notre exemple, la valeur retournée est contenue dans la variable `p` de type `double`. C'est pourquoi l'en-tête de la fonction s'écrit :

  ```
  public static double périmètre (int r)
  ```

  De cette façon, le compilateur sait, à la seule lecture de l'en-tête, que la fonction transmet un résultat de type `double`.

La fonction `périmètre()` s'écrit en résumé de la façon suivante :

```
public static double périmètre (int r)
{
  double p;
  p = 2 * Math.PI * r;
  return p;
}
```

Dans notre exemple, la fonction `périmètre()` utilise un seul paramètre et retourne un résultat numérique. Dans d’autres situations, le nombre de paramètres peut varier, et les fonctions peuvent avoir soit aucun, soit plusieurs paramètres. De la même façon, une fonction peut ne pas retourner de résultat.

Les fonctions au sein d’un programme Java

Avec les fonctions, nous voyons apparaître la notion de fonctions **appelées** et de programmes **appelant** des fonctions.

Dans notre exemple, la fonction `périmètre()` est appelée par la fonction `main()`. Cette dernière est considérée par l’ordinateur comme étant le programme principal (le terme anglais `main` se traduit par principal). En effet, la fonction `main()` est la première fonction exécutée par l’ordinateur au lancement d’un programme Java.

Toute fonction peut appeler ou être appelée par une autre fonction. Ainsi, rien n’interdit que la fonction `périmètre()`soit appelée par une autre fonction que la fonction `main()`.

Seule la fonction `main()` ne peut pas être appelée par une autre fonction du programme. En effet, la fonction `main()` n’est exécutée qu’une seule fois, et uniquement par l’interpréteur Java, lors du lancement du programme.

Comment placer plusieurs fonctions dans un programme

Les fonctions sont des programmes distincts les uns des autres. Elles sont en outre définies séparément les unes des autres. Pour exécuter un programme constitué de plusieurs fonctions, il est nécessaire, pour l’instant, de les regrouper dans un même fichier, une même classe.


Voir, au chapitre 7, *« Les classes et les objets »,* la section *« Compilation et exécution d’une application multifichier ».*

Pour des raisons pédagogiques, les fonctions `main()` et `périmètre()` ont été présentées séparément. En réalité, ces deux fonctions sont placées à l’intérieur de la même classe `Cercle` (*définie notamment au chapitre introductif, « Naissance d’un programme »*).

Le programme prend la forme suivante :

```
public class Cercle  // Le fichier s’appelle Cercle.java
{
public static void main(String [] parametre)
  {
   // Déclaration des variables
   int valeur ;
   double résultat ;
   System.out.print("Valeur du rayon : ") ;
   valeur = Lire.i() ;
   résultat = périmètre (valeur) ;
   System.out.print("rayon = " + valeur + " perimetre = " + résultat);
  } // fin de main()
```

```
  public static double périmètre (int r)
  {
    double p ;
    p = 2 * Math.PI * r ;
    return p ;
  } // fin de périmètre()
} //fin de class Cercle
```

En examinant la structure générale de ce programme, nous remarquons qu'il existe deux blocs d'instructions séparés, nommés `main()` et `périmètre()`. Ces deux blocs sont placés à l'intérieur d'un bloc représentant la classe `Cercle`, comme illustré à la Figure 5-1.

Figure 5–1.
Les fonctions `main()` *et* `périmètre()`, *à l'intérieur de la classe* `Cercle`.

```
public class Cercle
{
  public static void main(String [] arg)
  {

  }

  public static double périmètre (int r)
  {

  }
}
```

Nous observons que la structure de la fonction `périmètre()` est très voisine de celle de la fonction `main()`. Elle est constituée d'un en-tête, suivi d'un corps, formé d'un bloc défini par des accolades, ouvrante et fermante.

Notons, pour finir, que la fonction `main()` est ici placée avant la fonction `périmètre()` mais qu'il est aussi permis de l'écrire après. L'ordre d'apparition des fonctions dans une classe importe peu et est laissé au choix du programmeur.

Les différentes formes d'une fonction

Nous l'avons déjà observé (*voir la section « Principes de fonctionnement » de ce chapitre*), les fonctions peuvent posséder zéro, un, voire plusieurs paramètres de différents types. De la même façon, elles peuvent fournir ou non un résultat. Suivant les cas, leur définition varie légèrement.

Fonction avec résultat

Comme nous l'avons observé lors de la définition de la fonction `périmètre()`, toute fonction fournissant un résultat possède un `return` placé dans le corps de la fonction.

De plus, l'en-tête de la fonction possède obligatoirement un type, qui correspond au type du résultat retourné.

Si une fonction retourne en résultat une variable de type `int`, son en-tête s'écrit `public static int nomdelafonction()`.

Remarquons qu'une fonction ne retourne qu'une et une seule valeur. Il n'est donc pas possible d'écrire l'instruction `return` sous la forme `return a,b ;` pour retourner deux valeurs au programme appelant. Dans un tel cas, le compilateur détecte une erreur du type : « `',' expected` ».

Lorsqu'une fonction fournit plusieurs résultats, la transmission des valeurs ne peut se réaliser par l'intermédiaire de l'instruction `return`. Il est nécessaire dans ce cas d'employer des techniques plus avancées (*voir le chapitre 7, « Les classes et les objets »*).

Fonction sans résultat

Une fonction peut ne pas fournir de résultat. Tel est, en général, le cas des fonctions utilisées pour l'affichage de messages. Par exemple, la fonction `menu()` suivante ne fournit pas de résultat et ne fait qu'exécuter les opérations selon la valeur du paramètre `choix` :

```
public static void menu (int choix)
{
  switch (choix)
  {
   case 1 :
    // Saisie d'une personne
   break;
   case 2 :
    // Afficher une personne
   break;
  }
} // fin de menu()
```

L'en-tête `public static void menu (int choix)` mentionne que la fonction `menu()`ne retourne pas de résultat grâce au mot-clé `void` placé devant le nom de la fonction.

Si une fonction ne retourne pas de résultat, son en-tête est de type `void`, et l'instruction `return` ne figure pas dans le corps de la fonction.

Fonction à plusieurs paramètres

Prenons pour exemple une fonction `max()` qui fournit en résultat la plus grande des deux valeurs données en paramètres :

```
public class Maximum  // Le fichier s'appelle Maximum.java
{
  public static void main(String [] parametre)
  {
   // Déclaration des variables
   int v1, v2, sup;
   System.out.print("Entrer une valeur : ");
   v1 = Lire.i();
```

```
    System.out.print("Entrer une valeur : ");
    v2 = Lire.i();
    sup = max (v1,v2);
    System.out.print("Le max de " + v1 + " et de " + v2 + " est " + sup);
} // fin de main()

  public static int max (int a, int b)
  {
   int m = a;
   if (b > m) m = b;
   return m;
  } // fin de max()
} //fin de class Maximum
```

La fonction `max()`possède un en-tête :

```
public static int max (int a, int b)
```

qui mentionne deux paramètres, `a` et `b`, de type entier. Nous observons que :

- Lorsqu'une fonction possède plusieurs paramètres, ceux-ci sont séparés par une virgule. L'en-tête d'une fonction peut alors prendre la forme suivante :

  ```
  public static int quelconque (int a, char c, double t)
  ```

- Devant chaque paramètre est placé son type, même si deux paramètres consécutifs sont de type identique. Ainsi, écrire l'en-tête de la fonction `max()` de la façon suivante :

  ```
  public static int max (int a, b)
  ```

  n'est pas possible et provoque une erreur de compilation de type `Identifier expected`.

Fonction sans paramètre

Une fonction peut ne pas avoir de paramètre. Son en-tête ne possède alors aucun paramètre entre parenthèses.

Ainsi, la fonction `sortie()` suivante permet de sortir proprement de n'importe quel programme :

```
public static void sortie ()
{
 System.out.print("Au revoir et a bientot...");
// Fonction Java qui permet de sortir proprement d'un programme
 System.exit(0);
}
```

Résumé

Un algorithme paramétré est une marche à suivre qui fournit un résultat pouvant différer suivant la valeur du ou des paramètres. Dans le langage Java, les algorithmes paramétrés s'appellent des fonctions ou encore des méthodes.

Le langage Java propose un ensemble de fonctions prédéfinies fort utiles. Parmi ces fonctions, se trouvent les fonctions mathématiques, telles que `Math.sqrt()`, pour calculer la racine carrée du nombre placé entre parenthèses, ou `Math.log()`, pour le logarithme.

L’étude des fonctions mathématiques montre que :

- Pour exécuter une fonction, il est nécessaire d’écrire dans une instruction le nom de la fonction choisie, suivi des paramètres éventuels, placés entre parenthèses.
- Toute fonction possède, dans son nom d’appel, des parenthèses, ouvrante et fermante.
- Le type et l’ordre des paramètres dans l’appel de la fonction doivent être respectés, sous peine d’obtenir une erreur de compilation ou d’exécution.

Le langage Java offre en outre au programmeur la possibilité d’écrire ses propres fonctions, de façon à obtenir des programmes bien adaptés au problème qu’il doit résoudre. La définition d’une fonction passe par plusieurs étapes, qui permettent de :

- Préciser les instructions composant la fonction, en les plaçant dans le corps de la fonction. Ce dernier est déterminé par des accolades `{ }`.
- Associer le nom de la fonction aux instructions à l’aide d’un en-tête, qui précise le nom de la fonction, le type des paramètres (appelés paramètres formels) et le type de résultat retourné. Cet en-tête se rédige sous la forme suivante :

  ```
  public static type nomdelafonction (paramètres)
  ```

- Établir les `paramètres` utiles à l’exécution de la fonction en les déclarant à l’intérieur des parenthèses placées juste après le nom de la fonction.
  - Lorsqu’une fonction possède plusieurs paramètres, ceux-ci sont séparés par une virgule. Devant chaque paramètre est placé son type, même si deux paramètres consécutifs sont de type identique.
  - Lorsqu’une fonction n’a pas de paramètre, son en-tête ne possède aucun paramètre entre parenthèses.
- Préciser le type de résultat fourni par la fonction dans l’en-tête de la fonction et placer l’instruction `return` dès que le résultat doit être transmis au programme appelant la fonction.
  - Toute fonction fournissant un résultat possède un `return` placé dans le corps de la fonction.
  - L’en-tête de la fonction possède obligatoirement un type, qui correspond au type de résultat retourné. Remarquons qu’une fonction ne retourne qu’une et une seule valeur.
  - Si une fonction ne retourne pas de résultat, son en-tête est de type `void`, et l’instruction `return` ne figure pas dans le corps de la fonction.

Une fonction peut être appelée (exécutée) depuis une autre fonction ou depuis la fonction `main()`, qui représente le programme principal. L’appel d’une fonction est réalisé en écrivant une instruction composée du nom de la fonction suivi, entre parenthèses, d’une liste de paramètres.

Exercices

Apprendre à déterminer les paramètres d'un algorithme

5.1 Pour écrire l'algorithme permettant de réaliser une boisson plus ou moins sucrée, procédez de la façon suivante :

a. Écrivez le bloc d'instructions qui place un nombre déterminé de morceaux de sucre dans une boisson chaude.

b. Déterminez le paramètre qui permet de sucrer plus ou moins la boisson.

c. Donnez un nom à l'algorithme, et précisez le paramètre.

d. Écrivez l'algorithme en utilisant le nom du paramètre.

e. Appelez l'algorithme paramétré par son nom, en tenant compte du nombre de morceaux de sucre souhaité.

Comprendre l'utilisation des fonctions

5.2 À la lecture du programme suivant :

```
public class Fonction
{
  public static void main(String [] parametre)
  {
   // Déclaration des variables
   int a,compteur;
   for (compteur = 0; compteur <= 5; compteur++)
   {
     a = calculer(compteur);
     System.out.print(a + " a ");
   }
  } // fin de main()

  public static int calculer(int x)
  {
    int y;
    y = x * x;
    return y ;
  } // fin de fonc1()
} //fin de class
```

a. Délimitez les trois blocs définissant la fonction `main()`, la fonction `calculer()` et la classe `Fonction`.

b. Quel est le paramètre formel de la fonction `calculer()` ?

c. Quelles sont les valeurs transmises au paramètre de la fonction `calculer()` lors de son appel depuis la fonction `main()` ?

d. Quels sont les résultats produits par la fonction `calculer()` ?

e. Quelles sont les valeurs transmises à la variable `a` ?

f. Décrivez l'affichage réalisé par la fonction `main()`.

5.3 Soit la fonction :

```
public static int f( int x)
{
 int resultat;
 resultat = -x * x + 3 * x - 2;
 return resultat;
}
```

a. Écrivez la fonction `main()` qui affiche le résultat de la fonction `f(x)` pour `x = 0`.

b. Transformez la fonction `main()` de façon à calculer et à afficher le résultat de la fonction pour `x` entier variant entre `-5` et `5`. Utilisez pour cela, dans la fonction `main()`, une boucle `for` avec un indice variant entre `-5` et `5`.

c. Pour déterminer le maximum de la fonction `f(x)` entre `-5` et `5`, calculez la valeur de `f(x)` pour chacune de ces valeurs, et stockez le maximum dans une variable `max`.

Détecter des erreurs de compilation concernant les paramètres ou le résultat d'une fonction

5.4 Déterminez les erreurs de compilation des extraits de programmes suivants :

a. En utilisant la fonction `max()` décrite au cours de ce chapitre :

```
public static void main(String [] parametre)
{
 // Déclaration des variables
 double v1, v2, sup;
 System.out.print("Entrer une valeur : ");
 v1 = Lire.d();
 System.out.print("Entrer une valeur : ");
 v2 = Lire.d();
 sup = max (v1,v2);
 System.out.print("Le max de " + v1 + " et " + v2 + " est " + sup);
} // fin de main()
```

b.

```
public static int max (int a, int b)
{
    float m = a;
    if (m > b) m = b;
    return m;
} // fin de max()
```

c. En utilisant la fonction `menu()` décrite au cours de ce chapitre :

```
public static void main(String [] parametre)
{
// Déclaration des variables
   int v1, v2 ;
   System.out.print("Entrer une valeur : ");
   v1 = Lire.i();
   v1 = menu (v2);
} // fin de main()
```

d.

```
public static void menu (int c)
{
   switch (c)
   {...
   }
  return c;
}
```

Écrire une fonction simple

5.5 Écrivez la fonction `pourcentage()`, qui permet de calculer les pourcentages d'utilisation de la Carte Bleue, du chéquier et des virements automatiques, sachant que la formule de calcul du pourcentage pour la Carte Bleue est, comme nous l'avons vu au chapitre 1, « Stocker une information », la suivante :

Nombre de paiements par Carte Bleue / Nombre total de paiements * 100.

Suivez les étapes décrites dans le présent chapitre :

a. Déterminez les instructions composant la fonction.

b. Associez le nom de la fonction aux instructions.

c. Pour déterminer les paramètres de la fonction, recherchez les valeurs pouvant modifier le résultat du calcul.

✔ Aide : l'en-tête d'une fonction ayant deux paramètres entiers s'écrit :
`public static type nomdelafonction( int a, int b).`

d. Précisez le type de résultat fourni par la fonction.

e. Écrivez la fonction `main()` qui fait appel à la fonction `pourcentage()` et qui permette d'obtenir une exécution telle que :

```
Nombre de paiement par Carte Bleue : 5
Nombre de cheques emis : 10
Nombre de virements automatiques : 5
Vous avez emis 20 ordres de debit
dont 25.0 % par Carte Bleue
     50.0 % par cheque
     25.0 % par virement
```

Le projet « Gestion d'un compte bancaire »

Le programme écrit au chapitre 4, « Faire des répétitions », est suffisamment structuré pour y placer des fonctions. En effet, chaque option du projet est un programme à part entière et peut donc être décrite sous forme de fonction.

Dans le cadre de ce chapitre, nous allons construire trois fonctions relativement simples, qui vont nous permettre de comprendre le mécanisme de construction des fonctions.

Définir une fonction

Les fonctions sans paramètre avec résultat

La fonction `menuPrincipal()`affiche le menu principal du programme et demande la saisie de l'option choisie. Cette valeur doit être communiquée à la fonction `main()` pour exécuter la structure `switch` qui suit cette fonction.

a. Décrivez l'en-tête de la fonction `menuPrincipal()`, en prenant soin de préciser le type correspondant à la valeur retournée.

b. Placez les instructions relatives à l'affichage du menu et à la saisie de l'option dans le corps de la fonction.

c. Vérifiez que l'opérateur `return` soit appliqué à la variable contenant le choix de l'option.

Les fonctions sans paramètre ni résultat

La fonction `sortir()` affiche un message de politesse avant de sortir proprement du programme. Elle ne fournit pas de résultat et n'a pas non plus besoin de paramètre, puisque aucune valeur spécifique n'est nécessaire à son exécution.

a. Décrivez l'en-tête de la fonction `sortir()`.

b. Déterminez les instructions composant cette fonction, et placez-les dans le corps de la fonction.

La fonction `alAide()` affiche à l'écran une explication sur ce que réalise chaque option de l'application.

a. Décrivez l'en-tête de la fonction `alAide()`.

b. Déterminez les instructions composant cette fonction, et placez-les dans le corps de la fonction.

Appeler une fonction

Modifiez la fonction `main()` de votre programme de façon à utiliser les trois fonctions `alAide()`, `sortir()` et `menuPrincipal()`, définies aux étapes précédentes.

L'exécution finale du programme doit être identique à celle du chapitre précédent. Seule la structure interne du programme est modifiée, ce dernier étant composé de quatre « blocs fonctions ».

6

Fonctions, notions avancées

La création et l'utilisation de fonctions dédiées à la résolution d'un problème donné sont, nous l'avons observé au chapitre précédent, des opérations fondamentales, qui permettent le développement de logiciels dont le code source soit facilement réutilisable.

Ces fonctions transforment la structure générale des programmes et apportent, de ce fait, de nouveaux concepts, qu'il est important de bien maîtriser avant d'étudier la programmation objet.

Nous commençons par examiner (*section « La structure d'un programme »*), ces nouvelles notions, telles que la visibilité des variables, les variables locales et les variables de classe, à partir d'exemples simples. Pour chacune de ces notions, nous observons leur répercussion sur le résultat des différents programmes donnés en exemple.

Nous analysons ensuite (*section « Les fonctions communiquent »*), comment les fonctions échangent des données par l'intermédiaire des paramètres et du retour de résultat. À partir de cette analyse, nous constatons que ces modes de communication ne permettent pas toujours d'obtenir l'opération souhaitée.

La structure d'un programme

Nous avons déjà observé (*voir, au chapitre précédent, la section « Les fonctions au sein d'un programme Java »*) qu'un programme était constitué d'une classe, qui englobe un ensemble de fonctions définissant chacune un bloc d'instructions indépendant.

En réalité, il existe trois principes fondamentaux qui régissent la structure d'un programme Java. Ces principes sont détaillés ci-dessous.

1. Un programme contient :

- une fonction principale, appelée fonction `main()` ;
- un ensemble de fonctions définies par le programmeur ;

- des instructions de déclaration de variables.

2. Les fonctions contiennent :
   - des instructions de déclaration de variables ;
   - des instructions élémentaires (affectation, test, répétition, etc.) ;
   - des appels à des fonctions, prédéfinies ou non.
3. Chaque fonction est comparable à une boîte noire, dont le contenu n'est pas visible en dehors de la fonction.

De ces trois propriétés, découlent les notions de visibilité des variables, de variables locales et de variables de classe. Concrètement, ces trois notions sont attachées au lieu de déclaration des variables, comme l'illustre la Figure 6-1.

Figure 6–1.
Les variables peuvent être déclarées à l'intérieur ou à l'extérieur des fonctions mais toujours dans une classe.

```
public class NomDeLaClasse
{

  //Déclaration de variables

  public static void main(String [] arg)
  {
   //Déclaration de variables

   //Instructions élémentaires (if, for,...)
   //Appel de fonctions prédéfinies ou non
  }

  public static type nomFonction(paramètre)
  {
   //Déclaration de variables

   //Instructions élémentaires (if, for,...)
   //Appel de fonctions prédéfinies ou non
  }
}
```

Pour mieux comprendre ces différents concepts, nous allons observer un programme composé de deux fonctions, `main()` et `modifier()`, et d'une variable, nommée `valeur`. La fonction `modifier()` a pour objectif de modifier le contenu de la variable `valeur`. Pour chaque exemple, la variable `valeur` est déclarée en un lieu différent du programme. À partir de ces variations, le programme fournit un résultat différent, que nous analysons.

La visibilité des variables

Après étude des trois propriétés énoncées ci-dessus, nous observons qu'un programme est constitué de **déclarations de variables** et de **fonctions**. Il existe, de fait, une notion d'extérieur et d'intérieur aux fonctions. Les instructions élémentaires, de type affectation, test, etc., se situent toujours à l'intérieur d'une fonction, alors que la déclaration de variables est une opération réalisable à l'intérieur ou à l'extérieur d'une fonction.

De plus, la troisième propriété énumérée ci-dessus exprime qu'une fonction ne peut pas utiliser dans ses instructions une variable déclarée dans une autre fonction. Pour mieux visualiser cette propriété, examinons le programme ci-dessous.

Exemple : code source complet

```
public class Visibilite
{
  public static void main(String [] paramètre)
  {
   // Déclaration des variables
   int valeur = 2 ;
   System.out.println("Valeur = " + valeur + " avant modifier() ");
   modifier();
   System.out.println("Valeur = " + valeur + " apres modifier() ");
} // fin de main()

  public static void modifier ()
  {
   valeur = 3 ;
   System.out.println("Valeur = " + valeur + " dans modifier() ");
  } // fin de modifier
} //fin de class Visibilite
```

Dans ce programme, nous constatons que l'instruction `valeur = 3 ;`, placée dans la fonction `modifier()`, cherche à modifier le contenu de la variable `valeur`, déclarée non pas dans la fonction `modifier()` mais dans la fonction `main()`.

```
public class Visibilite
{
  public static void main(String [] arg)
  {
   //Déclaration de variables
     int valeur = 2 ;
  }
                                        valeur
                                          2
  public static void modifier()
  {
   valeur = 3;
  }
}
```

Figure 6–2.
Une variable déclarée dans une fonction ne peut pas être utlisisée par une autre fonction.

Cette modification n'est pas réalisable, car la variable `valeur` n'est définie qu'à l'intérieur de la fonction `main()`. Elle est donc **invisible** depuis la fonction `modifier()`. Les fonctions sont, par définition, des blocs distincts. La fonction `modifier()`ne peut agir sur la variable `valeur`, qui n'est visible qu'à l'intérieur de la fonction `main()`.

C'est pourquoi le fait d'écrire l'instruction `valeur = 3 ;` dans la fonction `modifier()` provoque une erreur de compilation du type : `Line 12 : Undefined variable : valeur`.

Variable locale à une fonction

La deuxième propriété énoncée précédemment établit qu'une fonction est formée d'instructions élémentaires, et notamment des instructions de déclaration de variables.

Par définition, une variable déclarée à l'intérieur d'une fonction est dite **variable locale à la fonction**. Pour l'exemple précédent, la variable `valeur` est locale à la fonction `main()`.

Les variables locales n'existent que pendant le temps de l'exécution de la fonction. Elles ne sont pas modifiables depuis une autre fonction. Nous l'avons vu à la section précédente, le contenu de la variable `valeur` ne peut être modifié par une instruction située en dehors de la fonction `main()`.

Cependant, le programmeur débutant qui souhaite modifier à tout prix la variable `valeur` va chercher à corriger, dans un premier temps, l'erreur de compilation énoncée ci-dessus. Pour cela, il déclare une variable `valeur` à l'intérieur de la fonction `modifier()` et une autre à l'intérieur de la fonction `main()`. De cette façon, la variable `valeur` est définie dans chacune des fonctions, et aucune erreur de compilation n'est détectée. Examinons plus précisément ce que réalise un tel programme.

Exemple : code source complet

```
public class VariableLocale
{
  public static void main(String [] paramètre)
  {
   // déclaration de variables locales
   int valeur = 2 ;
   System.out.println("Valeur = " + valeur + " avant modifier() ");
   modifier();
   System.out.println("Valeur = " + valeur + " apres modifier() ");
  } // fin de main()

  public static void modifier ()
  {
   // déclaration de variables locales
   int valeur ;
   valeur = 3 ;
   System.out.println("Valeur = " + valeur + " dans modifier() ");
  } // fin de modifier
} //fin de class VariableLocale
```

Pour bien comprendre ce qu'effectue ce programme, construisons le tableau d'évolution (*voir, au chapitre 1, « Stocker une information », la section « L'instruction d'affectation »*) de chaque variable déclarée dans le programme `Cercle.java`.

Puisque les fonctions `main()` et `modifier()` sont des blocs d'instructions séparés, l'interpréteur Java crée un emplacement mémoire pour chaque déclaration de la variable `valeur`. Il existe deux cases mémoire `valeur` distinctes portant le même nom. Elles sont

distinctes parce qu'elles n'appartiennent pas à la même fonction. Le tableau des variables déclarées pour chaque fonction est le suivant :

Variable locale à `main()`	valeur
`valeur = 2 ;`	2

Variable locale à `modifier()`	valeur
`valeur = 3 ;`	3

Résultat de l'exécution

L'exécution du programme a pour résultat :

```
Valeur = 2 avant modifier()
Valeur = 3 dans modifier()
Valeur = 2 apres modifier()
```

À l'exécution du programme, le premier appel à la fonction `System.out.println()` affiche le contenu de la variable `valeur` définie dans la fonction `main()`, soit `2`.

Le programme réalise ensuite les actions suivantes :

- Appeler la fonction `modifier()`, qui affiche le contenu de la variable `valeur` définie à l'intérieur de cette fonction, soit `3`.
- Sortir de la fonction `modifier()` et détruire la variable `valeur` locale à cette fonction.
- Retourner à la fonction `main()` et afficher de nouveau le contenu de la variable `valeur` définie dans la fonction `main()`, soit `2`.

Figure 6–3.

Toute variable déclarée à l'intérieur d'une fonction est une variable locale, propre à cette fonction.

```
public class VariableLocale
{
  public static void main(String [] arg)
  {
     //Déclaration de variables
     int valeur = 2 ;                      valeur: 2
  }
  public static void modifier()
  {
     //Déclaration de variables
     int valeur = 3 ;                      valeur: 3
  }
}
```

La variable `valeur` est déclarée deux fois dans chacune des deux fonctions, et nous constatons que la fonction `modifier()` ne change pas le contenu de la variable `valeur` déclarée dans la fonction `main()`. En réalité, même si ces deux variables portent le même nom, elles sont totalement différentes, et leur valeur est stockée dans deux cases mémoire distinctes.

En cherchant à résoudre une erreur de compilation, nous n'avons pas écrit la fonction qui modifie la valeur d'une variable définie en dehors d'elle-même. Cette modification est

impossible dans la mesure où la variable `valeur` n'est connue que de la fonction, et d'aucune autre.

Variable de classe

En examinant plus attentivement la première propriété définie au début de ce chapitre (*voir section « La structure d'un programme »*), nous remarquons que les classes contiennent également des instructions de déclaration, en dehors de toute fonction. Les variables ainsi déclarées sont appelée **variables de classe**. Elles sont définies pour l'ensemble du programme et sont visibles depuis toutes les fonctions.

La déclaration des variables de classe se réalise comme décrit ci-dessous.

Exemple : code source complet

```
public class VariableDeClasse
{
  // déclaration de variables de classe
  static int valeur ;
  public static void main(String [] paramètre)
  {
   valeur = 2 ;
   System.out.println("Valeur = " + valeur + " avant modifier() ");
   modifier();
   System.out.println("Valeur = " + valeur + " apres modifier() ");
} // fin de main()

  public static void modifier ()
  {
   valeur = 3 ;
   System.out.println("Valeur = " + valeur + " dans modifier() ");
  } // fin de modifier
} //fin de class VariableDeClasse
```


Figure 6-4.
Une variable déclarée en dehors de toute fonction est appelée variable de classe.

Grâce à l'instruction `static int valeur ;`, la variable `valeur` est définie pour tout le programme `VariableDeClasse`. Le mot-clé `static` est important, car lorsque l'interpréteur Java le rencontre, il créé une case mémoire en un seul exemplaire, accessible depuis n'importe quelle méthode (*voir, au chapitre 8, « Les principes du concept Objet », la section « La communication objet »*).

La représentation par blocs du programme (*voir Figure 6-4*) montre que la variable `valeur` est visible tout au long du programme.

Puisque la variable `valeur` est déclarée à l'extérieur des fonctions `main()` et `modifier()`, elle est définie comme étant une variable de la classe `VariableDeClasse`. La variable `valeur` existe tout le temps de l'exécution du programme, et les fonctions définies à l'intérieur de la classe peuvent l'utiliser et modifier son contenu.

Résultat de l'exécution

L'exécution du programme a pour résultat :

```
Valeur = 2 avant modifier()
Valeur = 3 dans modifier()
Valeur = 3 apres modifier()
```

La variable `valeur` étant une variable de classe, l'ordinateur ne crée qu'un seul emplacement mémoire. Le tableau d'évolution de la variable est le suivant :

Variable de classe	valeur
`valeur = 2 // dans la fonction main()`	2
`valeur = 3 // dans la fonction modifier()`	3
`valeur = 3 // dans la fonction main()`	3

Puisqu'il n'existe qu'une seule case mémoire nommée `valeur`, celle-ci est commune à toutes les fonctions du programme, qui peuvent y déposer une valeur. Lorsque la fonction `modifier()` place `3` dans la case mémoire `valeur`, elle écrase la valeur `2`, que la fonction `main()` avait précédemment placée.

En utilisant le concept de variable de classe, nous pouvons écrire une fonction qui modifie le contenu d'une variable définie en dehors de la fonction.

Quelques précisions sur les variables de classe

Puisque les variables locales ne sont pas modifiables depuis d'autres fonctions et que, à l'inverse, les variables de classe sont vues depuis toutes les fonctions du programme, le programmeur débutant aura tendance, pour se simplifier la vie, à n'utiliser que des variables de classe.

Or, l'utilisation abusive de ce type de variables comporte plusieurs inconvénients, que nous détaillons ci-dessous.

Déclarer plusieurs variables portant le même nom

L'emploi systématique des variables de classe peut être source d'erreurs, surtout lorsqu'on prend l'habitude de déclarer des variables portant le même nom. Observons le programme suivant :

```
public class MemeNom
{
 // déclaration de variables de classe
 static int valeur ;
  public static void main(String [] paramètre
  {
   valeur = 2 ;
   System.out.println("Valeur = " + valeur + " avant modifier() ");
   modifier();
   System.out.println("Valeur = " + valeur + " apres modifier() ");
   } // fin de main()

  public static void modifier ()
  {
   System.out.println(valeur + " dans modifier() avant la déclaration");
   // Déclaration de variables locales
   int valeur ;
   valeur = 3 ;
   System.out.println(valeur + " dans modifier() apres la declaration");
  } // fin de modifier
} //fin de class MemeNom
```

Dans ce programme, la variable `valeur` est déclarée deux fois, une fois comme variable de classe et une autre fois comme variable locale à la fonction `modifier()`. Rien n'interdit de déclarer plusieurs fois une variable portant le même nom dans des blocs d'instructions différents.

Le fait de déclarer deux fois la même variable n'est cependant pas sans conséquence sur le résultat du programme.

Dans la fonction `modifier()`, les deux variables `valeur` coexistent et représentent deux cases mémoire distinctes. Lorsque l'instruction `valeur = 3` est exécutée, l'interpréteur Java ne peut placer la valeur numérique `3` dans les deux cases mémoire à la fois. Il est obligé de choisir. Dans un tel cas, la règle veut que ce soit la variable locale qui soit prise en compte et non la variable de classe.

Le résultat final du programme est le suivant :

```
Valeur = 2 avant modifier()
2 dans modifier() avant la declaration
3 dans modifier() après la dclaration
Valeur = 2 après modifier()
```

La modification n'est valable que localement. Lorsque le programme retourne à la fonction `main()`, la variable locale n'existe plus. Le programme affiche le contenu de la variable de classe, soit `2`.

Le véritable nom d'une variable de classe

Une variable de classe se différencie des variables locales par son nom. Lorsqu'une variable de classe est déclarée, l'ordinateur lui donne un nom, qui lui permet de la distinguer des autres variables.

Ce nom est constitué du nom de la classe, suivi d'un point puis du nom de la variable déclarée. Pour l'exemple suivant, la variable de classe `valeur` s'appelle en fait `VeritableNom.valeur`. Le programme peut s'écrire de la façon suivante :

```
public class VeritableNom
{
  // déclaration de variables de classe
  static int valeur ;
  public static void main(String [] paramètre)
  {
   VeritableNom.valeur = 2 ;
   System.out.println(VeritableNom.valeur + " avant modifier() ");
   modifier();
   System.out.println(VeritableNom.valeur + " apres modifier() ");
  } // fin de main()

  public static void modifier ()
  {
   System.out.println("Variable de classe : " + VeritableNom.valeur );
   // Déclaration de variables locales
   int valeur = 3 ;
   System.out.println("Variable locale : " + valeur );
   VeritableNom.valeur = 3 ;
   System.out.println("Variable de classe : " + VeritableNom.valeur );
  } // fin de modifier
} //fin de class VeritableNom
```

En écrivant la variable de classe par son nom véritable, l'ambiguïté sur l'emploi de la variable de classe ou de la variable locale est levée, et l'exécution du programme a le résultat suivant :

```
2 avant modifier()
Variable de classe : 2
Variable locale : 3
Variable de classe : 3
3 après modifier()
```

Pour éviter toute méprise, il est recommandé d'utiliser les variables de classe avec parcimonie et chaque fois avec leur nom complet. En pratique, seules les variables qui présentent un intérêt général pour le programme sont à déclarer comme variables de classe.

De l'indépendance des fonctions

Comme nous l'avons déjà observé (*voir, au chapitre précédent, la section « Algorithme paramétré »*), une fonction est avant tout un sous-programme indépendant, capable d'être exécuté autant de fois que nécessaire et traitant des données différentes.

En construisant des fonctions qui utilisent des variables de classe, nous créons des fonctions qui ne sont plus des modules de programmes indépendants mais des extraits de programmes travaillant tous sur le même jeu de variables.

Cette dépendance aux variables de classe nuit au programme, car il est nécessaire, pour réutiliser de telles fonctions, de modifier tous les noms des variables de classe de façon à les rendre compatibles avec les nouveaux programmes.

En cas de développement de logiciels importants, comportant des centaines de milliers d'instructions, la transformation et l'amélioration des fonctionnalités du programme se trouvent fortement compromises. L'ensemble du code doit être examiné précisément afin de déterminer où se trouve la variable de classe concernée par la transformation envisagée.

Dans ce cadre, il convient de prendre les règles suivantes :

- Utiliser les variables de classe en nombre limité, le choix de ce type de variable s'effectuant en fonction de l'importance de la variable dans le programme. Une variable est considérée comme une variable de classe lorsqu'elle est commune à un grand nombre de fonctions.
- Écrire un programme de façon modulaire, chaque fonction travaillant de façon indépendante, à partir de valeurs transmises à l'aide des techniques étudiées à la section suivante.

Les fonctions communiquent

L'emploi systématique des variables de classe peut être, comme nous venons de le voir, source d'erreurs. Pour limiter leur utilisation, il existe des techniques simples, qui font que deux fonctions communiquent le contenu d'une case mémoire locale de l'une des fonctions à une case mémoire locale de l'autre.

Ces techniques sont basées sur le paramétrage des fonctions et sur le retour de résultat.

✔ Voir, au chapitre 5, *« De l'algorithme paramétré à l'écriture de fonctions »,* la section *« Les différentes formes d'une fonction ».*

Pour mieux cerner le fonctionnement de chacune de ces techniques, nous allons les étudier à l'aide d'un programme composé de deux fonctions, `main()` et `tripler()`, et d'une variable, `valeur`, locale à la fonction `main()`. La fonction `tripler()` a pour objectif de multiplier par trois le contenu de la variable `valeur`.

Le passage de paramètres par valeur

Notre contrainte est cette fois de n'utiliser que des variables locales. Nous supposons donc que la variable `valeur` soit locale à la fonction `main()`. Pour multiplier par trois cette valeur, la fonction `tripler()` doit connaître effectivement le contenu de la variable `valeur`.

La fonction `main()` doit communiquer pour cela le contenu de la variable `valeur` à la fonction `modifier()`. Cette communication est réalisée en passant le contenu de la variable au paramètre de la fonction `tripler()`. Examinons le programme ci-dessous.

Exemple : code source complet

```
public class ParValeur
{
  public static void main (String [] paramètre)
  {
   // Déclaration des variables
   int valeur = 2 ;
   System.out.println("Valeur = " + valeur + " avant tripler() ");
   tripler(valeur);
   System.out.println("Valeur = " + valeur + " apres tripler() ");
   } // fin de main()

  public static void tripler (int valeur)
  {
   System.out.println("Valeur = " + valeur + " dans tripler() ");
    valeur = 3 * valeur;
   System.out.println("Valeur = " + valeur + " dans tripler() ");
  } // fin de tripler
} //fin de class ParValeur
```

Dans ce programme, deux variables `valeurs` sont déclarées. La première est locale à la fonction `main()`, tandis que la seconde est locale à la fonction `tripler()`. Cependant, comme la seconde est déclarée dans l'en-tête de la fonction, elle est considérée comme variable locale à la fonction et, surtout, comme paramètre formel de la fonction `tripler()`.

> ✔ Voir, au chapitre 5, *« De l'algorithme paramétré à l'écriture de fonctions »,* la section *« Définir une fonction ».*

De cette façon, lorsque la fonction `tripler()` est appelée depuis la fonction `main()`, avec, comme valeur de paramètre, le contenu de `valeur`, soit `2`, la variable `valeur` locale de `tripler()` prend la valeur `2` (*voir Figure 6-5*).

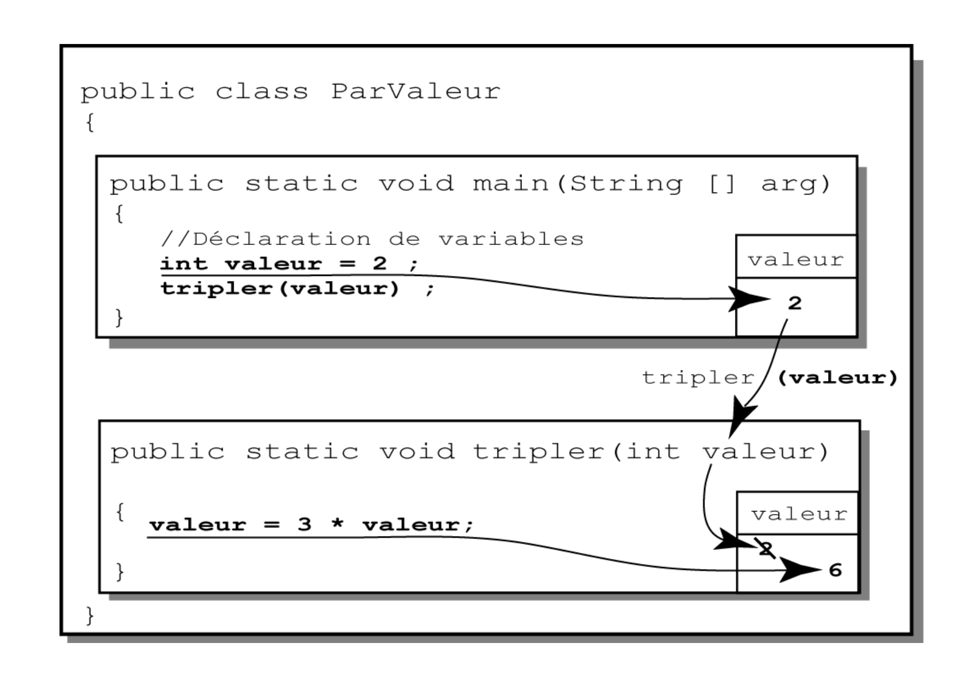


Figure 6-5.
Grâce au paramètre, le contenu d'une variable locale à la fonction appelante, `main()`, *est transmis à la fonction appelée,* `tripler()`.

Ensuite, la variable `valeur` locale à la fonction `tripler()` est multipliée par trois grâce à l'instruction `valeur = 3 * valeur;`. La variable `valeur` vaut donc `6` dans la fonction `tripler()`. Lorsque le programme sort de la fonction `tripler()` et retourne à la fonction `main()`, il détruit la variable locale de la fonction `tripler()` et affiche le contenu de la variable `valeur` locale à la fonction `main()`, soit encore `2`.

Résultat de l'exécution

```
Valeur = 2 avant tripler()
Valeur = 2 dans tripler()
Valeur = 6 dans tripler()
Valeur = 2 apres tripler()
```

Grâce au paramètre de la fonction `tripler()`, le contenu de la variable `valeur` locale à la fonction `main()` est transmis à la fonction `tripler()`. Une fois la fonction exécutée, nous constatons que la variable `valeur` de la fonction `main()` n'est pas modifiée pour autant.

En effet, la valeur passée en paramètre est copiée dans la case mémoire associée au paramètre. Même si le paramètre porte le même nom que la variable, il s'agit de deux cases mémoire distinctes. La modification reste donc locale à la fonction.

Lorsqu'une fonction communique le contenu d'une variable à une autre fonction par l'intermédiaire d'un paramètre, on dit que le **paramètre est passé par valeur**. Ce type de transmission de données ne permet pas de modifier, dans la fonction appelante, le contenu de la variable passée en paramètre.

Le résultat d'une fonction

Pour garder le résultat de la modification du contenu d'une variable en sortie de fonction, une technique consiste à retourner la valeur calculée par l'intermédiaire de l'instruction `return`.

Examinons le programme ci-dessous, qui utilise cette technique.

Exemple : code source complet

```
public class Resultat
{
  public static void main (String [] paramètre)
  {
   // Déclaration des variables
   int valeur = 2 ;
   System.out.println("Valeur = " + valeur + " avant tripler() ");
   valeur = tripler(valeur);
   System.out.println("Valeur = " + valeur + " apres tripler() ");
  } // fin de main()

  public static int tripler (int v)
  {
   System.out.println("v = " + v + " dans tripler() ");
```

```
    int résultat = 3*v ;
    System.out.println("Resultat = " + résultat + " dans tripler() ");
    return résultat;
   } // fin de tripler

 } //fin de class Resultat
```

Ici, le contenu de la variable `valeur` est passé au paramètre `v` de la fonction `tripler()`. Puisque le paramètre formel (`v`) correspond à une case mémoire distincte de la variable effectivement passée (`valeur`), il est plus judicieux de le déclarer sous un autre nom d'appel que celui de la variable, de façon à ne pas les confondre. En général, et tant que cela reste possible, nous avons pour convention de donner comme nom d'appel du paramètre formel la première lettre du paramètre réel. Pour notre exemple, `valeur` est le paramètre réel. Le paramètre formel s'appelle donc `v`.

Une fois le calcul réalisé à l'intérieur de la fonction `tripler()`, la valeur résultante placée dans la variable `résultat` est transmise à la fonction `main()` qui a appelé la fonction `tripler()`. Cette transmission est réalisée grâce à l'instruction `return résultat;`. Le contenu du résultat est alors placé dans la variable `valeur` grâce au signe d'affectation `=`, comme l'illustre la Figure 6-6.

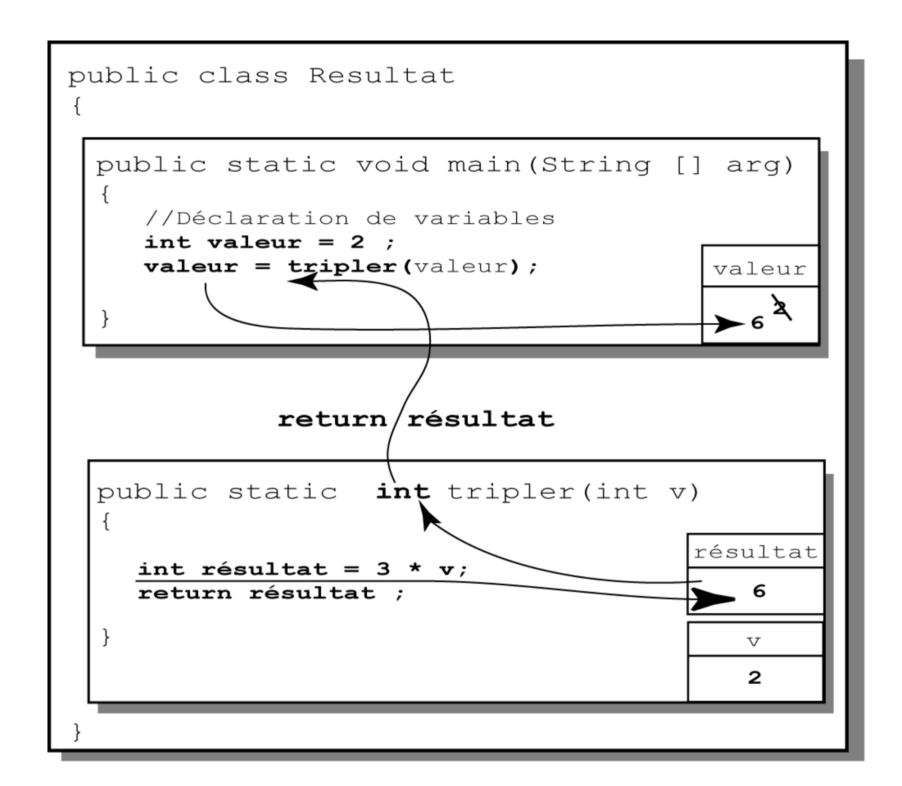


Figure 6-6.
Grâce au retour de résultat, le contenu d'une variable locale à la fonction appelée `tripler()` *est transmis à la fonction appelante* `main()`.

Résultat de l'exécution

```
Valeur = 2 avant tripler()
v = 2 dans tripler()
Resultat = 6 dans tripler()
Valeur = 6 après tripler()
```

Grâce à la technique du retour de résultat et du passage de paramètre par valeur, les fonctions peuvent échanger les contenus de variables. Les variables locales sont donc exploi-

tables aussi facilement que les variables de classe, tout en évitant les inconvénients liés à ces dernières.

Lorsqu'il y a plusieurs résultats à retourner

Une difficulté subsiste : le retour de résultat ne peut se réaliser que sur une seule valeur. Il n'est pas possible de retourner plusieurs valeurs à la fois. Si l'on souhaite écrire l'algorithme qui échange le contenu de deux variables (*voir, au chapitre 1, « Stocker une information », la section « Échanger les valeurs de deux variables »*) sous forme de fonction, nous nous trouvons confronté au problème décrit dans l'exemple ci-dessous.

Exemple : code source complet

```
public class PlusieursResultats
{
    public static void main (String [] arg)
    {
        int a, b;
        System.out.print("Entrer une valeur pour a : ");
        a = Lire.i();
        System.out.print("Entrer une valeur pour b : ");
        b = Lire.i();
        System.out.println(" a = "+a+" b = "+b);
        échange (a,b);
        System.out.println("Apres echange,");
        System.out.println(" a = "+a+" b = "+b);
    }

    public static void échange(int x, int  y)
    {
        int tmp = x;
        x = y;
        y = tmp;
    }
}
```

La fonction `échange()`réalise théoriquement l'échange du contenu des deux variables passées en paramètres. Si `a` prend la valeur `1` et que `b` vaille `2`, après exécution de la fonction `échange()` `a` doit prendre la valeur de `b`, soit `1`, et `b` la valeur de `a`, soit `2`.

Résultat de l'exécution

Examinons le résultat de l'exécution de ce programme, en supposant que les caractères grisés soient les valeurs choisies par l'utilisateur.

```
Entrer une valeur pour a : 1
Entrer une valeur pour b : 2
a = 1 b = 2
Apres echange,
a = 1 b = 2
```

Nous le constatons à l'exécution : aucun échange n'a été réalisé. Il n'y a rien d'étonnant à cela, puisque le passage des paramètres est un passage par valeur et qu'il ne modifie pas le contenu des paramètres réels `a` et `b` passés à la fonction `échange()`.

La solution qui consiste à retourner le résultat est impossible. En effet, il serait nécessaire de retourner les deux variables échangées, et il n'est pas possible d'écrire `return x, y ;`, la syntaxe de cette instruction n'étant pas valide.



Voir, au chapitre 5, *« De l'algorithme paramétré à l'écriture de fonctions »*, la section *« Les différentes formes d'une fonction »*.

Dans l'état actuel de nos connaissances, nous ne sommes pas à même de récupérer différentes valeurs modifiées au sein d'une fonction. En réalité, seul le concept d'objet permet de réaliser un tel exploit. Nous l'étudions au chapitre suivant.

Résumé

Un programme Java est structuré selon les trois principes fondamentaux suivants :

1. Un programme contient :
   - une fonction principale, appelée fonction `main()` ;
   - un ensemble de fonctions définies par le programmeur ;
   - des instructions de déclaration de variables.

2. Les fonctions contiennent :
   - des instructions de déclaration de variables ;
   - des instructions élémentaires (affectation, test, répétition, etc.) ;
   - des appels à des fonctions, prédéfinies ou non.

3. Chaque fonction est comparable à une boîte noire, dont le contenu n'est pas visible en dehors de la fonction.

De ces trois propriétés découlent les notions suivantes :

- **Visibilité** : toute variable déclarée à l'intérieur d'une fonction n'est visible que dans cette fonction et ne peut être utilisée dans une autre fonction.
- **Variable locale** : toute variable déclarée à l'intérieur d'une fonction est une variable locale à cette fonction. Ces variables n'existent que le temps de l'exécution de la fonction, et elles ne sont pas modifiables depuis une autre fonction.
- **Variable de classe** : les variables déclarées en dehors de toute fonction sont appelées des variables de classe. Ces variables sont définies pour l'ensemble du programme, et elles sont visibles et modifiables par toutes les fonctions de la classe.

Lorsqu'une variable de classe et une variable locale portant le même nom coexistent à l'intérieur d'une fonction, la règle veut que ce soit la variable locale qui soit prise en compte et non la variable de classe.

Les fonctions sont des blocs d'instructions distinctes. Pour communiquer le contenu d'une case mémoire (variable) locale de l'une à une case mémoire locale de l'autre fonction, il est nécessaire d'utiliser les techniques suivantes :

- **Les paramètres des fonctions** : lorsqu'une fonction communique le contenu d'une variable à une autre fonction par l'intermédiaire d'un paramètre, on dit que le paramètre est passé **par valeur**. Ce type de transmission de données ne permet pas de modifier, dans la fonction appelante, le contenu de la variable passée en paramètre.
- **Le retour de résultat** : pour garder en résultat la modification du contenu d'une variable en sortie de fonction, une technique consiste à retourner la valeur calculée par l'intermédiaire de l'instruction `return`.

Ces deux modes de communication ne permettent pas de récupérer plusieurs données modifiées à l'intérieur d'une fonction. Seul le concept d'objet, étudié au chapitre suivant, permet de réaliser cette opération.

Exercices

Repérer les variables locales et les variables de classe

6.1 En observant le programme suivant :

```
public class Calculette
{
 public static double résultat ;

 public static void main( String [] argument)
 {
  int a, b;
  menu();
  System.out.println("Entrer la premiere valeur ");
  a = Lire.i();
  System.out.println("Entrer la seconde valeur ");
  b = Lire.i();
  calculer();
  afficher();
 }

 public static void calculer()
 {
  char opération ;
  switch (opération)
  {
   case '+' : résultat = a + b ;
              break ;
   case '-' : résultat = a - b ;
              break ;
   case '/' : résultat = a / b ;
```

```
                break ;
      case '*' : résultat = a * b ;
                break ;
     }
   }

   public static void afficher()
   {
    char opération ;
    System.out.print(a + "  " +opération + "  "+ b + " =  " + résultat);
   }

   public static void menu()
   {
    char opération ;
    System.out.println("Je sais compter, entrez l'operation choisie") ;
    System.out.println(" + pour additionner ") ;
    System.out.println(" - pour soustraire ") ;
    System.out.println(" * pour pultiplier ") ;
    System.out.println(" / pour diviser ") ;
    System.out.println(" (+, -, *, /)  ? : ") ;
    opération = Lire.c() ;
   }
}
```

a. Recherchez les différentes fonctions définies dans la classe `Calculette`.

b. Dessinez le programme sous forme de schéma, en représentant les fonctions à l'aide de blocs. Placez les variables dans les blocs où elles sont déclarées.

c. À l'aide du schéma, déterminez les variables locales à chacune des fonctions, ainsi que les variables de classe.

d. Après exécution de la fonction `menu()` et lecture des deux valeurs numériques `a` et `b`, la fonction `calculer()` peut-elle réaliser l'opération demandée ? Pourquoi ?

e. Même question pour la fonction `afficher()`.

Communiquer des valeurs à l'appel d'une fonction

6.2 Pour corriger le programme `Calculette`, nous supposons que les variables `résultat` et `opération` soient déclarées en tant que variables de classe et non plus localement aux fonctions `afficher()` et `menu()`.

a. Modifiez le schéma réalisé en **6.1.b**, en tenant compte de ces nouvelles déclarations.

b. Quelle technique doit-on utiliser pour que les fonctions `calculer()` et `afficher()` connaissent le contenu des variables `a` et `b`, afin d'effectuer ensuite les instructions qui les composent ?

c. Écrivez les fonctions en utilisant cette technique.

Transmettre un résultat à la fonction appelante

6.3 Nous supposons que le programme `Calculette` ne contienne plus de variables de classe. Les variables `résultat` et `opération` sont maintenant déclarées localement aux fonctions qui les utilisent.

a. Quelles sont les conséquences de cette nouvelle hypothèse sur le résultat du programme ?

b. Comment la fonction `calculer()` peut-elle connaître l'opérateur choisi par l'utilisateur dans la fonction `menu()` ?

c. Transformez la fonction `menu()` de sorte que l'opérateur soit transmis à la fonction `main()`.

d. Modifiez la fonction `calculer()` de façon à lui transmettre l'opérateur fourni par la fonction `menu()`.

e. Comment la fonction `afficher()` peut-elle connaître le résultat de la fonction `calculer()` ?

f. Transformez la fonction `calculer()` de sorte que le résultat soit transmis à la fonction `main()`.

g. Modifiez la fonction `afficher()` de façon à lui transmettre le résultat fourni par la fonction `calculer()`.

Le projet « Gestion d'un compte bancaire »

Au chapitre précédent, nous avons construit trois fonctions, `alAide()`, `sortir()` et `menuPrincipal()`, qui améliorent la lisibilité du programme. Ces fonctions concernent surtout l'affichage de messages de dialogue de l'application vers l'utilisateur (menu, aide, etc.). Elles réalisent l'interface entre l'utilisateur et l'application sans transformer les données propres à chaque compte bancaire.

Pour réaliser les opérations de création et d'affichage d'un compte (options `1` et `2` du menu), nous allons ici construire des fonctions qui modifient, transforment les données d'un compte.

Comprendre la visibilité des variables

La fonction `afficherCpte()` réalise l'option `2` du menu principal de notre application. Cette fonction affiche l'ensemble des caractéristiques d'un compte, soit son numéro, son type, son taux, s'il s'agit d'un compte d'épargne, et sa valeur courante. Nous supposons, que l'ensemble de ces valeurs aient été préalablement saisies en `option 1`.

Les variables locales

Une première solution pourrait s'écrire

```
public static void afficherCpte()
{
  long num ;
  char type ;
```

```
double taux ;
double val ;
System.out.print("Le compte n° : " + num + " est un compte ");
  if (type == 'C') System.out.println(" courant  ");
  else if (type == 'J') System.out.println(" joint  ");
  else if (type == 'E')
  {
       // affiche son taux dans le cas d'un compte d'épargne.
       System.out.println(" epargne  dont le taux est  " + taux);
  }
  System.out.println(" Valeur initiale : " + val);
}
```

- Quelles valeurs sont affichées par cette fonction ? Pourquoi ?

Les variables de classe

Pour corriger la fonction précédente, il est nécessaire que la fonction ait accès aux valeurs stockées lors de l'option 1.

Une première solution consiste à définir les variables à afficher comme variables de classe.

a. Transformez votre programme, et déclarez les variables `num`, `type`, `taux` et `val` comme variables de classe.

b. Retirez les déclarations des variables `num`, `type`, `taux` et `val` dans la fonction `afficherCpte()` de façon à éviter qu'elles soient encore utilisées par l'interpréteur comme variables locales.

c. Exécutez votre programme, et vérifiez que la fonction affiche correctement les valeurs.

Le passage de paramètres par valeur

Une seconde solution revient à déclarer les variables `num`, `type`, `taux` et `val` en paramètres de la fonction d'affichage, de façon à transmettre les valeurs saisies depuis la fonction `main()` (option 1) à la fonction `afficherCpte()`.

a. Décrivez l'en-tête de la fonction `afficherCpte()`, en prenant soin de déclarer en paramètre une variable pour chaque caractéristique du compte à transmettre à la fonction.

b. Déterminez les instructions composant cette fonction, et placez-les dans le corps de la fonction.

Les limites du retour de résultat

La fonction `créerCpte()` rassemble les instructions de l'option 1, soit l'affichage de messages et la saisie au clavier des valeurs caractéristiques d'un compte.

a. Recherchez quel doit être le résultat de la fonction à transmettre à la fonction `main()`.

b. Pour décrire l'en-tête de la fonction `créerCpte()`, est-il possible de déterminer le type à placer dans l'en-tête de la fonction ? Pourquoi ?

7

Les classes et les objets

L'étude du chapitre 6, « Fonctions, notions avancées », montre que, si une fonction fournit plusieurs résultats, ceux-ci ne peuvent pas être transmis au programme appelant. Pour contourner cette difficulté, il est nécessaire d'utiliser des objets, au sens de la programmation objet.

Pour faire comprendre les principes fondamentaux de la notion d'objet, nous étudions (*section « La classe* `String`*, une approche vers la notion d'objet »*), comment définir et gérer des objets de type `String`. Ce type permet la représentation des mots en tant que suites de caractères. À partir de cette étude, nous analysons les instructions qui font appel aux objets `String` afin d'en comprendre les principes de notation et d'utilisation.

Nous examinons ensuite (*section « Construire et utiliser ses propres classes »*) comment définir de nouveaux types de données. Pour cela, nous déterminons les caractéristiques syntaxiques d'une classe et observons comment manipuler des objets à l'intérieur d'une application et comment utiliser les méthodes qui leurs sont associées.

La classe `String`, une approche vers la notion d'objet

La classe `String` est une classe prédéfinie du langage Java. Elle permet de définir des « variables » contenant des suites de caractères, autrement dit des mots, ou, dans le jargon informatique, des **chaînes de caractères**. Nous étudions comment définir ces « variables » à la section ci-dessous.

La classe `String` est un type de données composé d'un grand nombre d'outils, ou méthodes, qui facilitent l'utilisation des chaînes de caractères (*voir la section « Les différentes méthodes de la classe* `String` *»*).

Manipuler des mots en programmation

L'utilisation des chaînes de caractères apporte beaucoup à la convivialité des programmes informatiques. Il serait impensable aujourd'hui de créer un logiciel de gestion du personnel sans pouvoir définir le nom et le prénom de chaque employé. Dans le même ordre d'idée, que serait la recherche d'informations sur Internet sans ces fameuses chaînes de caractères ?

Grâce aux chaînes de caractères, nous oublions le langage binaire, et il devient aisé de communiquer avec l'ordinateur dans notre propre langue. Pourtant, l'utilisation de ces fameuses chaînes a longtemps été source de difficultés.

Les mots nécessitent un type de données particulier, du fait qu'un mot possède, par nature, un nombre quelconque de caractères. À la différence des formats `int`, `double` ou `char`, les chaînes de caractères ne peuvent, *a priori*, être représentées par un nombre fixe de cases mémoire.

Déclaration d'une chaîne de caractères

Tout comme nous déclarons des variables pour stocker des valeurs entières ou réelles, nous devons déclarer une variable pour mémoriser la suite des caractères d'un mot ou d'une phrase. Le type de cette variable est le type `String`.

Le type `String` n'est pas un type simple, puisqu'il permet de regrouper sous un seul nom de variable plusieurs données, c'est-à-dire l'ensemble des caractères d'un mot.

Pour éviter les difficultés liées à la variation du nombre de caractères dans un mot, le langage Java fixe la longueur du mot en fonction de sa déclaration. Cela fait, le contenu du mot ne peut plus être modifié. En déclarant un objet de type `String`, il est possible, en même temps, de l'initialiser en lui affectant des caractères placés entre guillemets.

La déclaration suivante permet de créer un objet appelé `mot`, qui contient la chaîne de caractères `"exemple"` :

```
String mot = "exemple";
```

Remarquons que la variable `mot` n'est pas un ensemble de sept cases mémoire contenant les sept caractères du mot `exemple`. Lors de la déclaration de la variable `mot`, l'interpréteur Java crée une case qui contient l'adresse de la case où se trouve le premier caractère du mot `exemple` (*voir Figure 7-1*).

Lorsque l'ordinateur souhaite afficher la variable `mot`, il va rechercher l'information se situant à l'adresse stockée dans la case mémoire `mot`. On dit alors que la variable `mot` **pointe** sur la case qui contient la suite de caractères.

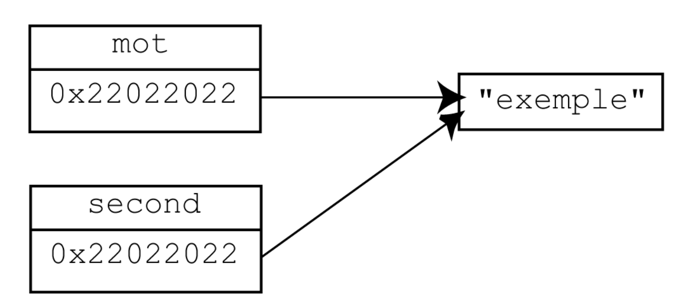


Figure 7–1.
Seul un objet de type `String` *contenant le mot* `"exemple"` *existe.* `mot` *et* `second` *font tous deux référence à cet objet unique.*

Les variables de type `String` ne contiennent pas directement l'information qui les caractérise mais seulement l'adresse où trouver cette information. Dès lors, ces variables ne s'appellent plus des variables mais des objets.

Les objets, au sens de la programmation objet, ne sont pas des « variables » de type simple (`int`, `long`, `double`, `char`, etc.). Ils correspondent à un type qui permet de regrouper plusieurs données sous une même adresse.

Lorsqu'un objet `second` est déclaré comme ci-dessous, il contient la même adresse (référence) que l'objet `mot`.

```
String second = mot;
```

Si le programme modifie le contenu de l'objet `mot` en lui affectant, par exemple, une nouvelle chaîne, l'interpréteur ne modifie pas la case pointée par `mot`, dans la mesure où, par définition, le contenu d'un mot ne peut être modifié.

```
mot = "nouveau";
```

Il crée en réalité une nouvelle adresse et lui associe la nouvelle chaîne de caractères. Pour notre exemple, l'objet `mot` est associé à la chaîne de caractères `'nouveau'`, et `second` reste associé à `'exemple'`.

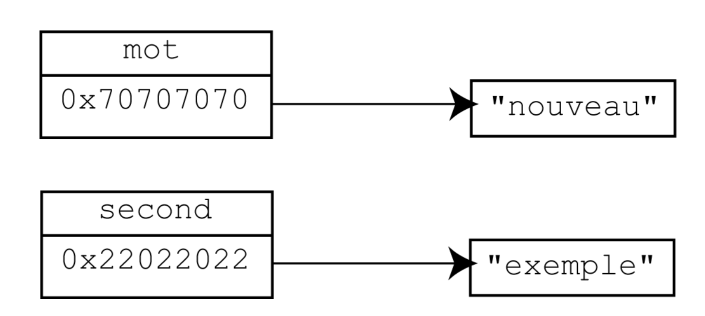


Figure 7–2.
La modification de `mot` entraîne la création d'une nouvelle chaîne de caractères et d'une nouvelle référence, automatiquement attribuées à `mot`. L'objet `second` conserve la précédente référence.

Les différentes méthodes de la classe `String`

L'utilisation des mots dans un programme est aujourd'hui incontournable. Il ne s'agit certes pas simplement d'afficher des mots mais de les traiter de la façon la plus intelligente possible. Ces traitements sont, par exemple, le tri alphabétique ou encore la recherche de mots particuliers dans un texte.

Pour réaliser ces opérations, la langage Java propose un ensemble de méthodes prédéfinies. Les méthodes d'une classe sont comparables aux fonctions, mais la terminologie « objet » les appelle **méthodes**

Ces méthodes offrent la possibilité de traiter rapidement et simplement l'information textuelle. Nous décrivons ci-dessous, regroupées par thème, une grande partie des méthodes définies dans la classe `String`. Nous donnons en exemple, pour chaque thème, un programme qui utilise ces méthodes.

Recherche de mots et de caractères

Opération	Fonction Java
Recherche si le mot se termine par le ou les caractères passés en paramètres.	`endsWith()`.
Recherche si le mot commence par le ou les caractères passés en paramètres.	`startsWith()`.
Recherche le caractère placé à la position spécifiée en paramètre. Le premier caractère occupe la position `0` et le dernier la position `length()-1` (*voir ci-dessous la description de* `length()`).	`charAt()`.
Localise un caractère ou une sous-chaîne dans un mot, à partir du début du mot. Renvoie la valeur –1 si le caractère ou la chaîne recherché ne fait pas partie du mot.	`indexOf()`.
Localise un caractère ou une sous-chaîne dans un mot à partir de la fin du mot. Renvoie la valeur –1 si le caractère ou la chaîne recherché ne fait pas partie du mot.	`lastIndexOf()`.
Extrait une sous-chaîne d'un mot.	`substring()`.

Exemple de recherche de mots et de caractères

```
public class Rechercher {
 public static void main(String [] argument)
 {
  String phrase = "Mieux vaut tard que jamais";
  String soumo ="";
  int place;
  System.out.println("Vous avez dit : " + phrase);
  soumo = phrase.substring(11,15);
  System.out.println("De 11 a 15, la sous chaine est : " + soumo);
  for ( int i = 0; i < 5; i++)
   System.out.println("en " + i + ", il y a : " + phrase.charAt(i));

  System.out.println("Entrez un mot : ");
  soumo = Lire.S();

  if(phrase.endsWith(soumo))
   System.out.println("La phrase se termine avec : " + soumo);
  else
   System.out.println("La phrase ne finit pas avec : " + soumo);

  place = phrase.indexOf(soumo);
  if (place == -1)
   System.out.println("Ce mot n'existe pas dans : " + phrase);
  else
   System.out.println(soumo+" est a la position " + place);
 }
}
```

Résultat de l'exécution

Exécution 1

```
Vous avez dit : Mieux vaut tard que jamais
De 11 a 15, la sous chaine est : tard
En 0, il y a : M
En 1, il y a : i
En 2, il y a : e
En 3, il y a : u
En 4, il y a : x
Entrez un mot : tard
La phrase ne finit pas avec : tard
tard est a la position : 11
```

`Phrase` et `soumo` sont deux objets de type `String`, initialisés respectivement à `"Mieux vaut tard que jamais"` et `" "` (mot ne comportant pas de caractère).

L'instruction `soumo = phrase.substring(11,15);` recherche la sous-chaîne située entre les caractères `11` et `15` de l'objet `phrase`. Cela fait, elle place l'ensemble de ces caractères dans l'objet `soumo`.

Grâce à l'instruction `phrase.charAt(i)`, placée dans l'instruction d'affichage `System.out.print`, le programme affiche les cinq premiers caractères de l'objet `phrase`, `i` variant de `0` à `4`.

Ensuite, `phrase.endsWith(soumo)` permet de savoir si l'objet `phrase` se termine avec la suite de caractères saisie au clavier et stockée dans l'objet `soumo`. Le résultat de la méthode `endsWith()` est `true` si la chaîne se termine par l'argument et `false` dans le cas contraire. Pour notre exemple, `soumo` vaut `"tard"`, et la méthode retourne `false`. Le programme exécute donc l'instruction placée dans le bloc `else` associé au test `if(phrase .endsWith(soumo))`.

Pour finir, l'instruction `phrase.indexOf(soumo);` recherche si l'objet `soumo` est contenu dans l'objet `phrase`. Si tel est le cas, elle retourne la position du premier caractère trouvé, sinon elle retourne `–1`. Ici, `tard` est détecté dans `"mieux vaut tard que jamais"` en position 11.

Exécution 2

```
Vous avez dit : Mieux vaut tard que jamais
De 11 a 15, la sous chaine est : tard
En 0, il y a : M
En 1, il y a : i
En 2, il y a : e
En 3, il y a : u
En 4, il y a : x
Entrez un mot : mais
La phrase se termine avec : mais
mais est a la position : 22
```

Si l'utilisateur saisit `"mais"` au lieu de `"tard"`, le test `if (phrase.endsWith(soumo))` est vrai, la méthode `endsWith()`retournant `true`. Le programme exécute donc l'instruction placée dans le bloc `if`.

Exécution 3

```
Vous avez dit : Mieux vaut tard que jamais
De 11 a 15, la sous chaine est : tard
En 0, il y a : M
En 1, il y a : i
En 2, il y a : e
En 3, il y a : u
En 4, il y a : x
Entrez un mot : OK
La phrase ne finit pas avec : OK
Ce mot n'existe pas dans : Mieux vaut tard que jamais
```

Si l'utilisateur saisit `"OK"` au lieu de `"mais"`, le test `if (phrase.endsWith(soumo))` est faux, et la méthode `endsWith()` retourne `false`. Le programme exécute l'instruction placée dans le bloc `else`. De plus, l'instruction `phrase.indexOf(soumo);` retourne –1 car `"OK"` n'est pas détecté dans `"mieux vaut tard que jamais"`. Le programme exécute alors le bloc `else` associé.

Comparaison de mots

Opération	Fonction Java
Compare deux mots et retourne une valeur : • Nulle si les deux mots sont identiques. • Positive si le premier mot est plus grand (placé après) le deuxième mot (dans le dictionnaire). • Négative si le premier mot est plus petit (placé avant) le deuxième mot (dans le dictionnaire).	`compareTo()`
Compare la valeur de deux mots. Elle retourne `true` si les deux chaînes sont identiques et `false` dans le cas contraire.	`equals()`
Compare la valeur de deux mots sans différencier les majuscules des minuscules. Elle retourne `true` si les deux chaînes sont identiques et `false` dans le cas contraire.	`equalsIgnoreCase()`
Détermine si deux portions de chaînes sont identiques. Dans l'affirmative, elle renvoie `true`.	`regionMatches()`

Exemple de comparaison de mots

```
public class Comparer
 {
  public static void main(String [] argument)
  {
  String prvb1 = "Le mieux est l'ennemi du bien";
  String prvb2 ="Le Mieux Est l'Ennemi du bien";
```

```
    System.out.println("1 : " + prvb1);
    System.out.println("2 : " + prvb2);

    System.out.println("Comparons les 10 premiers caracteres : ");

    System.out.print("En tenant compte des majuscules : ");
    if (prvb1.regionMatches(false, 0 ,prvb2 ,0 , 10))
     System.out.println("Les 10 premiers cars sont identiques");
    else
     System.out.println("Il y a des differences sur les 10 premiers cars");

    System.out.print("Sans tenir compte des majuscules : ");
    if (prvb1.regionMatches(18, prvb2, 18, 6))
     System.out.println("Les cars de 18 a 24 sont identiques");
    else
     System.out.println("Il y a des differences");

    if (prvb1.compareTo(prvb2) == 0)
     System.out.println("Les deux chaines sont identiques");
    else
    {
     if (prvb1.compareTo(prvb2) < 0)
       System.out.print(prvb1 + " est avant " + prvb2);
     else
       System.out.print(prvb1 + " est apres " + prvb2);
     System.out.println("dans le dictionnaire");
    }
    System.out.print("Sans tenir compte des majuscules : ");
    if (prvb1.equalsIgnoreCase(prvb2))
      System.out.println("Les deux chaines sont identiques");
     else
      System.out.println("Les deux chaines sont differentes");
  }
}
```

Résultat de l'exécution

```
1 : Le mieux est l'ennemi du bien
2 : Le Mieux Est l'Ennemi du bien
Comparons les 10 premiers caracteres :
En tenant compte des majuscules : Il y a des differences sur les 10 premiers
cars
Sans tenir compte des majuscules : Les cars de 18 a 24 sont identiques
Le mieux est l'ennemi du bien est apres Le Mieux Est l'Ennemi du bien dans
le dictionnaire;
Sans tenir compte des majuscules : Les deux chaines sont identiques
```

Les objets `prvb1` et `pvrb2` sont initialisés respectivement à `"Le mieux est l'ennemi du bien"` et `Le "Mieux Est l'Ennemi du bien"`.

La méthode `regionMatches()` s'utilise soit avec quatre paramètres, soit avec cinq paramètres. Dans ce programme, nous donnons en exemple les deux appels possibles :

- Le premier appel à la méthode utilise cinq paramètres (`regionMatches(false, 0, prvb2, 0, 10)`). Le premier paramètre est un booléen, qui, s'il est égal à `false`, permet de réaliser la comparaison des deux mots, en tenant compte de la présence des majuscules. Pour notre cas, la méthode détermine si les deux portions de chaîne `prvb1` et `pvrb2` (correspondant au troisième paramètre de la méthode) sont identiques, en tenant compte des majuscules.

  Cette recherche est réalisée à partir de la valeur spécifiée par le deuxième paramètre (soit `0`, c'est-à-dire le premier caractère de `prvb1`). Le quatrième paramètre représente la position du premier caractère à comparer dans l'objet `prvb2`. Le cinquième est le nombre de caractères consécutifs à comparer. Pour notre exemple, le programme recherche s'il y a des similitudes entre `prvb1` et `prvb2`, à partir du début des deux mots, et ce sur les dix caractères suivants.
- Le deuxième appel à la méthode est composé de quatre paramètres (`regionMatches(18, prvb2, 18, 6)`). En fait, ces quatre paramètres correspondent aux quatre derniers paramètres de l'appel décrit précédemment. Le booléen figurant dans l'appel précédant n'existe plus, car, par défaut, cette méthode travaille sans tenir compte des majuscules. Elle est donc équivalente à l'appel de la méthode suivante : `prvb1.regionMatches(true, 18, prvb2, 18, 6).`

Ensuite, l'instruction `prvb1.compareTo(prvb2)` compare les objets `prvb1` et `prvb2` et détermine s'ils sont identiques ou placés avant ou après dans l'ordre alphabétique.

Pour finir, l'instruction `prvb1.equalsIgnoreCase(prvb2)` vérifie si les deux objets `prvb1` et `prvb2` sont identiques ou non, sans tenir compte de la présence des majuscules.

Transformation de formats

Opération	Fonction Java
Transforme en minuscules la chaîne sur laquelle la méthode est appliquée.	`toLowerCase()`
Transforme en majuscules la chaîne sur laquelle la méthode est appliquée.	`toUpperCase()`
La méthode place (concatène) la chaîne spécifiée en paramètre à la suite de la chaîne sur laquelle la méthode est appliquée.	`concat()`
Remplace systématiquement dans la chaîne sur laquelle la méthode est appliquée tous les caractères donnés en premier argument par le caractère donné en deuxième argument.	`replace()`
Calcule le nombre de caractères de la chaîne sur laquelle la méthode est appliquée.	`length()`

Exemple de transformation de format

```
public class Transformer
{
  public static void main(String [] argument)
  {
```

```
    String phrase = "Qui dort ";
    String verbe = "dine";
    String p1 = "", p2 = "", p3 = "", p4 = "";
    int nbcar;
    System.out.println("1 : " + phrase);
    System.out.println("2 : " + verbe);
    p1 = phrase.toUpperCase();
    System.out.println("En majuscules : " + p1);
    p2 = phrase.toLowerCase();
    System.out.println("En minuscules : " + p2);
    p3 = phrase.concat(verbe);
    nbcar = p3.length();
    System.out.print("Apres concat() : ");
    System.out.println(p3 +" possede : " + nbcar + " caracteres");
    p4 = p3.replace('i','a');
    System.out.println("Apres replace() : " + p3 + " devient : " + p4);
  }
}
```

Résultat de l'exécution

```
1 : Qui dort
2 : dine
En majuscules : QUI DORT
En minuscules : qui dort
Apres concat() : Qui dort dine possede : 13 caracteres
Apres replace() : Qui dort dine devient : Qua dort dane
```

Les objets `phrase` et `verbe` sont initialisés respectivement à `"Qui dort"` et `"dine"`. L'instruction `p1 = phrase.toUpperCase();` transforme en majuscules le contenu de `phrase` et place cette transformation dans l'objet `p1`.

L'instruction `p2 = phrase.toLowerCase()` place dans `p2` le contenu de `phrase` transformé en minuscules. Notons que, pour chacune de ces instructions, l'objet `phrase` n'est jamais modifié.

L'instruction `p3 = phrase.concat(verbe);` place en bout de l'objet `phrase` le mot contenu dans `verbe`. Cela fait, le résultat de cette opération est affecté à l'objet `p3`. L'objet `phrase` n'est pas modifié.

Ensuite, l'instruction `nbcar = p3.length();` calcule la longueur de l'objet `p3`, c'est-à-dire le nombre de caractères constituant l'objet p3.

Pour finir, l'instruction `p4 = p3.replace('i', 'a');` remplace tous les caractères `'i'` de `p3` par des `'a'` et place le résultat de cette transformation dans l'objet `p4`. L'objet `p3` n'est pas modifié.

Appliquer une méthode à un objet

L'observation des exemples précédents montre que l'appel d'une méthode de la classe `String` ne s'écrit pas comme une simple instruction d'appel à une méthode (fonction), telle que nous l'avons étudiée jusqu'à présent.

Comparons l'appel à une méthode de la classe `Math` à celui d'une méthode de la classe `String`.

Par exemple, pour calculer la valeur absolue d'une variable `x`, les instructions sont les suivantes :

```
double x = 4, y;
y = Math.abs(x) ;
```

Pour transformer un mot en lettres majuscules, les instructions sont :

```
String mot = "petit", MOT;
MOT =  mot.toUpperCase();
```

Comme nous le constatons, dans le premier cas, la fonction `Math.abs()` s'applique à la variable `x`, en passant la valeur de `x` en paramètre. En effet, les variables `x` et `y` ne sont pas des objets au sens de la programmation objet. Elles sont de type `double` et représentent simplement le nom d'une case mémoire dans laquelle l'information est stockée. Aucune méthode, aucun traitement ne sont associés à cette information.

Dans la seconde écriture, la méthode `toUpperCase()` est appliquée à l'objet `mot` par l'intermédiaire d'un point (`.`), placé entre le nom de l'objet et la méthode. Les objets `mot` et `MOT` ne peuvent être considérés comme des variables. Ils sont de type `String`. L'information représentée par ce type n'est pas simple. Elle représente (*voir Figure 7-3*) les éléments suivants :

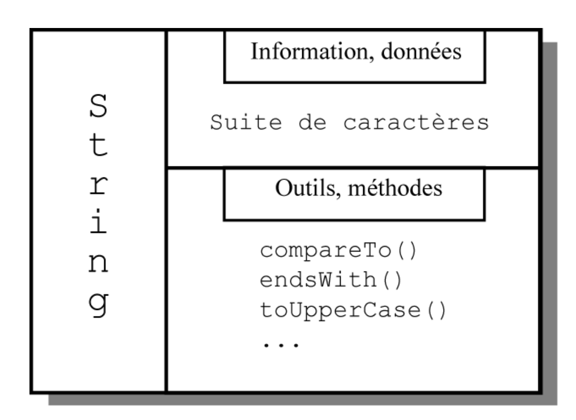


Figure 7–3.
La classe `String` *définit l'association de données et de méthodes applicables à ces données.*

- D'une part, une référence (une adresse) vers un ensemble de caractères stockés dans plusieurs cases mémoire distinctes.
- D'autre part, un ensemble de méthodes propres qui lui sont applicables. Ces méthodes sont l'équivalent d'une boite à outils, qui opère uniquement sur les objets de type `String`.

Quelle qu'elle soit, une classe correspond à un type, qui spécifie une association de données (informations ou valeurs de tout type) et de méthodes (outils d'accès et de transformation des données). Ces méthodes, définies à l'intérieur d'une classe, ne peuvent s'appliquer qu'aux données de cette même classe.

Grâce à cette association, une classe permet la définition de nouveaux types de données, qui structurent l'information à traiter (*voir, dans ce chapitre, la section « Construire et utiliser ses propres classes »*).

Principes de notation

À cause de cette différence fondamentale de représentation de l'information, l'emploi des méthodes à travers les objets utilise une syntaxe particulière.

Pour un objet de type `String`, cette syntaxe est la suivante :

```
// Déclaration et initialisation
String objet = "";
// La méthode s'applique à objet
objet.nomDeLaMéthode(liste des paramètres éventuels) ;
```

Pour appliquer une méthode à un objet, il suffit de placer derrière le nom de l'objet un point suivi du nom de la méthode et de ces paramètres. Remarquons que, par convention :

- Tout nom de méthode commence par une minuscule.
- Si le nom de la méthode est composé de plusieurs mots, ceux-ci voient leur premier caractère passer en majuscule.
- Le nom d'une classe commence toujours par une majuscule.

Grâce à cette écriture, l'objet est associé à la méthode, de façon à pouvoir modifier l'information (les données) contenue dans l'objet. Cette technique permet de récupérer les différentes données modifiées localement par une méthode. Elle est le principe de base du concept d'objet, décrit et commenté au chapitre suivant.

Construire et utiliser ses propres classes

L'étude de la classe `String` montre qu'une classe correspond à un type de données. Ce type est composé de données et de méthodes exploitant ces données. La classe `String` est un type prédéfini du langage Java.

Il existe d'autres types prédéfinis de classes dans le langage Java. Ces classes sont des outils précieux et efficaces, qui simplifient le développement des applications. Différentes classes sont examinées dans la troisième partie de cet ouvrage.

L'intérêt des classes réside aussi dans la possibilité de définir des types structurés, propres à un programme. Grâce à cette faculté, le programme se développe de façon plus sûre, les objets qu'il utilise étant définis en fonction du problème à résoudre.

Avant d'étudier réellement l'intérêt de la programmation objet et ses conséquences sur les modes de programmation (*voir le chapitre 8, « Les principes du concept d'objet »*), nous examinons dans les sections qui suivent comment créer des types spécifiques et utiliser les objets associés à ces nouveaux types.

Définir une classe et un type

Définir une classe, c'est construire un type structuré de données. Avant de comprendre les avantages d'une telle construction, nous abordons ici la notion de type structuré (et donc de classe) d'un point de vue syntaxique.

Pour définir un type, il suffit d'écrire une classe, qui, par définition, est constituée de données et de méthodes (*voir Figure 7-3*). La construction d'une classe est réalisée selon les deux principes suivants :

1. **Définition des données** à l'aide d'instructions de déclaration de variables et/ou d'objets. Ces variables sont de type simple, tel que nous l'avons utilisé jusqu'à présent (`int`, `char`, etc.) ou de type composé, prédéfini ou non (`String`, etc.).

   Ces données décrivent les informations caractéristiques de l'objet que l'on souhaite définir. Elles sont aussi appelées communément champ, attribut ou membre de la classe.

2. **Construction des méthodes** définies par le programmeur. Ce sont les méthodes associées aux données. Elles se construisent comme de simples fonctions, composées d'un en-tête et d'instructions, comme nous l'avons vu aux chapitres précédents.

   Ces méthodes représentent tous les traitements et comportements de l'objet que l'on cherche à décrire.

En définissant de nouveaux types, nous déterminons les caractéristiques propres aux objets que l'on souhaite programmer. Un type d'objet correspond à l'ensemble des données traitées par le programme, regroupées par thème.

Un objet peut être une `personne`, si l'application à développer gère le personnel d'une société, ou un `livre`, s'il s'agit d'une application destinée à la gestion d'une bibliothèque. Remarquons que l'objet `personne` peut aussi être utilisé dans le cadre d'un logiciel pour bibliothèque, puisqu'un lecteur empruntant un `livre` est aussi une `personne`.

Construire un type `Cercle`

Examinons, sur un exemple simple, la démarche de construction d'un type structuré. Observons pour cela comment construire le type de données qui décrive au mieux la représentation d'un cercle quelconque.

Cette réalisation passe par deux étapes : « Rechercher les caractéristiques propres à tout cercle » et « Définir le comportement de tout cercle ».

Rechercher les caractéristiques propres à tout cercle

D'une manière générale, tout cercle est défini grâce à son rayon. Si l'on souhaite afficher ce cercle, il est en outre nécessaire de connaître sa position à l'écran. Pour simplifier, nous supposons que la position d'un cercle soit déterminée grâce aux coordonnées de son centre.

Les caractéristiques d'un cercle sont son rayon et sa position à l'écran, c'est-à-dire les coordonnées en `x` (abscisse) et en `y` (ordonnée) du centre du cercle. Ces trois données sont représentables à l'aide de valeurs numériques, que nous choisissons, pour simplifier, de type `int`.

Pour déclarer les données d'un cercle, nous écrivons les déclarations suivantes :

```
public int x, y; // position du centre du cercle
public int r ;   // rayon
```

Définir le comportement de tout cercle

D'un point de vue informatique, plusieurs opérations peuvent être appliquées à un cercle. Un cercle peut être déplacé ou agrandi (*voir les méthodes* `déplacer()` *et* `agrandir()` *dans le code source ci-dessous*). Ces opérations modifient la valeur du rayon ou des coordonnées du centre du cercle à l'écran.

C'est pourquoi il est nécessaire de définir une méthode qui affiche à l'écran les données (rayon, position) d'un cercle avant ou après transformation (*voir la méthode* `afficher()`*dans le code source ci-dessous*).

La méthode de calcul du périmètre d'un cercle peut être utile (*voir la méthode* `périmètre()` *dans le code source ci-dessous*).

La classe descriptive du type `Cercle`

```
public class Cercle
{
 public int x, y;   // position du centre
 public int r;      // rayon

 public void afficher() //Affichage des données de la classe
 {
  System.out.println(" Cercle centre en " + x + ", " + y);
  System.out.println(" de rayon : " + r);
 }

 public double périmètre() //Calcul du périmètre d'un cercle
 {
  return 2*Math.PI*r;
 }

 public void déplacer(int nx, int ny) // Déplace le centre du cercle en
 {                                    // (nx, ny). Ces coordonnées étant
  x = nx;                             // passées en paramètres de la fonction
  y = ny;
 }

 public void agrandir(int nr)         // Augmente la valeur courante du
 {                                   // rayon avec la valeur passée en paramètre
  r = r + nr;
 }
} // Fin de la classe Cercle
```

La classe `Cercle`, décrite à l'intérieur d'un fichier appelé `Cercle.java`, définit un type de données composé de trois attributs caractéristiques des cercles, à savoir la position du centre en abscisse et ordonnée et le rayon, ainsi que quatre comportements différents. Sa description par bloc est représentée à la Figure 7-4.

Figure 7–4.

Les données `x`, `y` *et* `r` *du type* `Cercle` *sont déclarées en dehors de toute fonction. N'importe quelle modification de ces données est donc visible par l'ensemble des méthodes de la classe.*

```
public class Cercle
{
  public int x, y;
  public int r;

  public void agrandir(int nr)
  {
    r = nr;
  }

  public double périmètre()
  {
    return 2*Math.PI*r
  }

  public void affiche()
  {
    //...
  }

  public void déplacer(int nx, int ny)
  {
    //...
  }
}
```

x	y	r

Quelques observations

Suivant la description de la Figure 7-4, nous constatons que les données `x`, `y` et `r` sont déclarées en dehors de toute fonction. Par conséquent, chaque méthode a accès à tout moment aux valeurs qu'elle contient, soit pour les consulter, soit pour les modifier.

Les méthodes `afficher()` et `périmètre()` ne font que consulter le contenu des données `x`, `y` et `r` pour les afficher ou les utiliser en vue d'obtenir un nouveau résultat.

Au contraire, les méthodes `déplacer()` et `agrandir()` modifient le contenu des données `x`, `y` et `r`. Ces modifications, réalisées à l'intérieur d'une méthode, sont aussi visibles depuis les autres méthodes de la classe.

Il existe donc deux types de méthodes, les méthodes qui permettent d'accéder aux données de la classe et celles qui modifient ces données.

> ✔ Voir, au chapitre 8, «*Les principes du concept d'objet*», la section «*Les méthodes d'accès aux données*».

En comparant les programmes construits aux chapitres précédents à celui-ci, nous constatons les deux différences fondamentales suivantes :

- Le mot-clé `static` a disparu de toutes les instructions de déclaration. Cette disparition n'est pas sans conséquence sur le déroulement du programme. Elle permet de créer non plus de simples variables mais des objets (*voir, au chapitre 8, «Les principes du concept d'objet», la section «Les données* `static`»).
- Une classe définissant un type structuré ne possède pas de fonction `main()`. La définition d'une classe n'est pas la même chose que la réalisation d'une application. Une classe est une entité à part entière, qui définit globalement de quoi est constitué un objet et précise les opérations qu'il est possible de lui appliquer.

Bien entendu, une classe est définie pour être utilisée dans un programme exécutable (une application) qui contient une fonction `main()`. Nous abordons plus en détail cette opération à la section suivante.

Définir un objet

Après avoir défini un nouveau type structuré, l'étape suivante consiste à écrire une application qui utilise effectivement un « objet » de ce type. Pour cela, le programmeur doit déclarer les objets utiles à l'application et faire en sorte que l'espace mémoire nécessaire soit réservé.

Déclarer un objet

Cette opération simple s'écrit comme une instruction de déclaration, avec cette différence que le type de la variable n'est plus un type simple prédéfini mais un type structuré, tel que nous l'avons construit précédemment. Ainsi, dans :

```
// Déclaration d'un objet chose
TypeDeL'Objet chose ;
```

`TypeDeL'Objet` correspond à une classe définie par le programmeur. Pour notre exemple, la déclaration d'un cercle `A` est réalisée par l'instruction :

```
Cercle A ;
```

Cette déclaration crée une case mémoire, nommée `A`, destinée à contenir une référence vers l'adresse où sont stockées les informations concernant le cercle `A`. À ce stade, aucune adresse n'est encore déterminée.

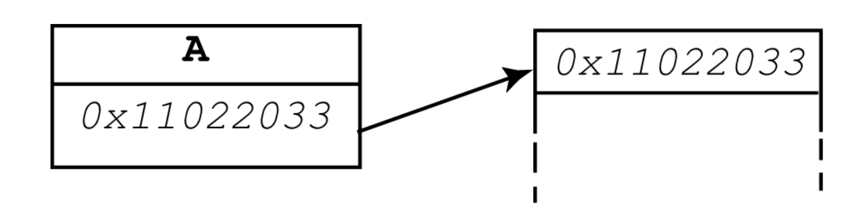


Figure 7–5.
La déclaration d'un objet réserve une case mémoire destinée à contenir l'adresse mémoire où seront stockées les informations. L'espace mémoire et l'adresse ne sont pas encore réservés pour réaliser ce stockage.

Réserver l'espace mémoire à l'aide de l'opérateur `new`

À cette étape, les informations caractérisant l'objet `A` ne peuvent être stockées, car l'espace mémoire servant à ce stockage n'est pas encore réservé. C'est l'opérateur `new` qui réalise cette réservation.

L'opérateur `new` est un programme Java, qui gère de lui-même la réservation de l'espace mémoire. Lorsqu'on applique cet opérateur à un objet, il détermine combien d'octets lui sont nécessaires pour stocker l'information contenue dans la classe.

Cet opérateur s'applique en écrivant à la suite du terme `new` le nom du type de l'objet déclaré, suivi de deux parenthèses.

```
// Réserver de l'espace mémoire pour l'objet chose
chose = new TypeDeL'Objet();
```

Pour notre exemple, la réservation de l'espace mémoire pour définir le cercle `A` s'écrit :

```
A = new Cercle() ;
```

Remarquons qu'il est possible de déclarer et de réserver de l'espace mémoire en une seule instruction :

```
Cercle A = new Cercle() ;
```

En écrivant une telle instruction, nous observons que, pour chaque objet déclaré, l'opérateur `new` réserve suffisamment d'espace mémoire pour stocker les données de la classe et pour copier les méthodes associées. Il détermine aussi l'adresse où sera stocké l'ensemble de ces informations (*l'espace mémoire pour l'objet* `A` *est illustré à la Figure 7-6*).

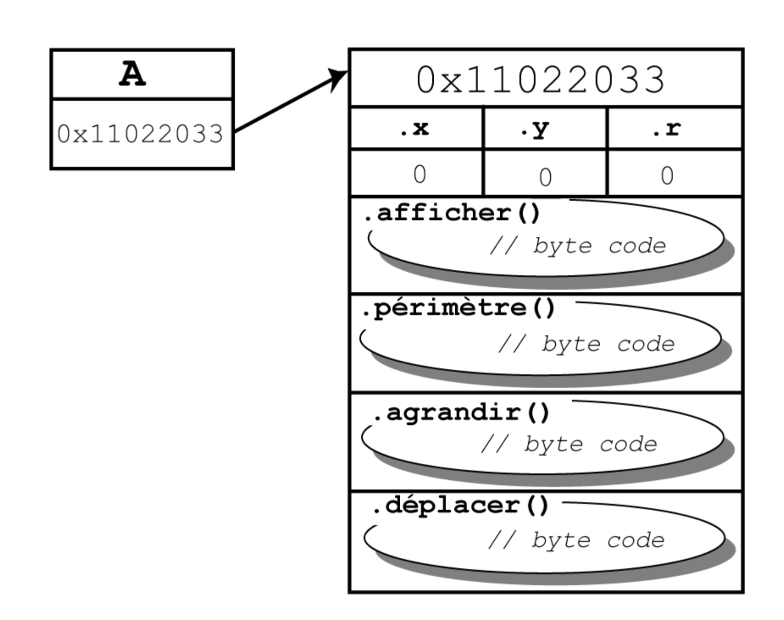


Figure 7–6.
Pour chaque objet créé, l'opérateur `new` *réserve un espace mémoire suffisamment grand pour y stocker les données et les méthodes descriptives de la classe. L'adresse est alors déterminée.*

Lors de cette réservation, l'interpréteur initialise les données de la classe à `0` pour les entiers, à `0.0` pour les réels, à `'\0'` pour les `char` et à `null` pour les `String`. Pour notre exemple, `A` est un cercle de rayon nul centré en (0, 0).

L'objet ainsi défini est un représentant particulier de la classe, caractérisé par l'ensemble de ses données. Dans le jargon informatique, on dit que l'objet `A` est une **instance** de la classe `Cercle`. Les données qui le caractérisent, à savoir `x`, `y` et `r`, sont appelées des **variables d'instance**.

Une instance est donc, en mémoire, un programme à part entière, composé de variables et de fonctions. Sa structure est telle qu'il ne peut s'exécuter et se transformer (c'est-à-dire modifier ses propres données) qu'à l'intérieur de cet espace. C'est pourquoi il est considéré comme une entité indépendante, ou « objet ».

Manipuler un objet

L'objet ainsi défini est entièrement déterminé par ses données et ses méthodes. Il est dès lors possible de modifier les valeurs qui le caractérisent et d'exploiter ses méthodes.

Accéder aux données de la classe

Pour accéder à une donnée de la classe de façon à la modifier, il suffit d'écrire

```
// Accéder à un membre de la classe
chose.nomDeLaDonnée = valeur du bon type ;
```

en supposant que le champ `nomDeLaDonnée` soit défini dans la classe correspondant au type de l'objet `chose`.

Pour notre exemple, la saisie au clavier des valeurs caractérisant le cercle `A` s'écrit de la façon suivante :

```
System.out.println(" Entrez la position en x : ") ;
A.x = Lire.i() ;
System.out.println(" Entrez la position en y : ") ;
A.y = Lire.i() ;
System.out.println(" Entrez le rayon : ");
A.r = Lire.i();
```

Les cases mémoire représentant les variables d'instance (`x`, `y` et `r`) de l'objet `A` sont accessibles *via* l'opérateur `point` (`.`).

Accéder aux méthodes de la classe

Pour appliquer une méthode de la classe à un objet particulier, la syntaxe utilise le même principe de notation :

```
// appliquer une méthode à l'objet chose
chose.nomDeLaMéthode(liste des paramètres éventuels) ;
```

en supposant que la méthode ait préalablement été définie pour le type de l'objet `chose`. Pour notre exemple, l'application de la méthode `périmètre()` à l'objet `A` s'écrit :

```
double p = A.périmètre();
```

Une application qui utilise des objets `Cercle`

L'exemple suivant montre comment exploiter, dans une application, l'ensemble des données et des méthodes définies dans la classe `Cercle`.

Exemple : code source complet

```
public class FaireDesCercles
{
 public static void main(String [] arg)
 {
  Cercle A  = new Cercle();
  A.affiche();
  System.out.println(" Entrez la position en x : ") ;
  A.x = Lire.i() ;
  System.out.println(" Entrez la position en y : ") ;
  A.y = Lire.i() ;
  System.out.println(" Entrez le rayon : ");
  A.r = Lire.i();
```

```
    A.affiche();

    double p = A.périmètre();
    System.out.println(" Votre cercle a pour perimetre : " + p);
    A.déplacer(5, 10);
    System.out.println(" Apres deplacement : ");
    A.affiche();
    A.agrandir(10);
    System.out.println(" Apres agrandissement : ");
    A.affiche();
  }
}
```

Compilation et exécution d'une application multifichiers

L'application `FaireDesCercles`, décrite dans le fichier `FaireDesCercles.java`, utilise le type `Cercle`, défini dans le fichier `Cercle.java`. Deux fichiers distincts sont donc nécessaires à la définition d'un programme qui utilise des objets `Cercle`.

Bien que cela puisse paraître curieux pour un débutant, l'application `FaireDesCercles` s'exécute correctement, malgré cette séparation des fichiers. Examinons comment fonctionne l'ordinateur dans un tel cas.

Nous l'avons déjà observé (*voir, au chapitre introductif, « Naissance d'un programme », la section « Exécuter un programme »*), deux phases sont nécessaires pour exécuter un programme Java : la phase de compilation et la phase d'interprétation. Si l'application est conçue avec plusieurs fichiers, ces deux phases sont aussi indispensables.

Phase de compilation

Lors de la compilation d'un programme constitué de plusieurs fichiers, la question se pose de savoir comment compiler l'ensemble de ces fichiers.

Pour simplifier la tâche de la personne qui développe des applications, le compilateur Java est construit de façon que seul le programme qui contient la fonction `main()` soit à compiler.

Au cours de la compilation, le compilateur constate de lui-même, au moment de la déclaration de l'objet, que l'application utilise des objets d'un type non prédéfini par le langage Java.

À partir de ce constat, il recherche, dans le répertoire où se trouve l'application qu'il compile, le fichier dont le nom corresponde au nouveau type qu'il vient de détecter et dont l'extension soit java. Tout programme Java a pour nom le nom de la classe (du type) qu'il définit.

Pour notre exemple, en compilant l'application `FaireDesCercles` grâce à la commande :

```
javac FaireDesCercles.java
```

le compilateur détecte le type `Cercle`. Il recherche alors le fichier `Cercle.java` dans le répertoire où se trouve l'application.

- Si le compilateur trouve ce fichier, il le compile aussi. En fin de compilation, deux fichiers ont été traités, `FaireDesCercles.java` et `Cercle.java`. Si le compilateur ne détecte aucune erreur, le répertoire contient les fichiers correpondant au pseudo-code et qui ont pour nom `FaireDesCercles.class` et `Cercle.class`.
- S'il ne trouve pas le fichier `Cercle.java`, il provoque une erreur de compilation du type `Class Cercle not found`.

Pour corriger cette erreur, il est possible de spécifier au compilateur où il peut trouver le fichier recherché en définissant une variable d'environnement `classpath`. Cette variable indique au compilateur quels sont les répertoires susceptibles de contenir des programmes Java. Cette définition se réalise de façon différente suivant le système utilisé, PC, Macintosh ou station Unix (*voir, sur le CD-Rom, la section « Construire son environnement de travail »*).

Phase d'interprétation

Une fois le programme compilé, l'exécution du programme est réalisée grâce à l'interpréteur de la machine virtuelle Java (JVM), qui exécute le pseudo-code associé au programme contenant la fonction `main()`. Pour notre exemple, la commande est :

```
java FaireDesCercles
```

Lorsque l'interpréteur trouve, en cours d'exécution, la déclaration d'un objet de type non prédéfini, il recherche, par l'intermédiaire du chargeur de classe (un programme, aussi appelé *class loader*, défini dans la JVM), le pseudo-code associé au type de l'objet et défini dans un fichier dont l'extension est `.class`. Pour notre exemple, le chargeur de classe recherche le fichier `Cercle.class`. Une fois trouvé, il charge le code en mémoire pour l'exécuter.

Analyse des résultats de l'application

Au cours des sections précédentes, nous avons observé que tout objet déclaré contenait une adresse correspondant à l'adresse où sont stockées les informations relatives à cet objet. Pour accéder aux données et méthodes de chaque objet, il suffit de passer par l'opérateur « . ».

Grâce à cette nouvelle façon de stocker l'information, les transformations d'un objet par l'intermédiaire d'une méthode de sa classe sont visibles pour tous les objets de la même classe. Autrement dit, si une méthode fournit plusieurs résultats, ces modifications sont visibles en dehors de la méthode et pour toute l'application.

Pour mieux comprendre cette technique, examinons comment s'exécute le programme `FaireDesCercles`. Les valeurs grisées correspondent aux valeurs saisies par l'utilisateur.

```
Entrez la position en x : 10
Entrez la position en y : 10
Entrez le rayon : 5
```

Les valeurs saisies au clavier par l'utilisateur sont directement stockées en `A.x`, `A.y` et `A.r` grâce aux instructions `A.x = Lire.i()`, ...

```
Cercle centre en 10,10
de rayon : 5
```

La méthode `afficher()` est appliquée à l'objet `A` (`A.afficher()`). Elle consulte et affiche les données associées à cet objet, soit `10` pour `x` (en réalité `A.x`), `10` pour `y` (en réalité `A.y`) et `5` pour `A.z`.

```
Votre cercle a pour perimetre : 31.41592653589793
```

De la même façon, la méthode `périmètre()` est appliquée à l'objet `A` (`A.périmètre()`). L'expression `2*Math.PI * r`, définie dans la méthode, est donc calculée pour `r` (`A.r`) valant `5`.

```
Après deplacement :
Cercle centre en 5, 2
de rayon : 5
```

L'instruction `A.déplacer(5, 2)` passe les nouvelles coordonnées de la position du centre du cercle en paramètres. Les données `x` et `y` de l'objet `A` sont modifiées en conséquence (*voir Figure 7-7*).

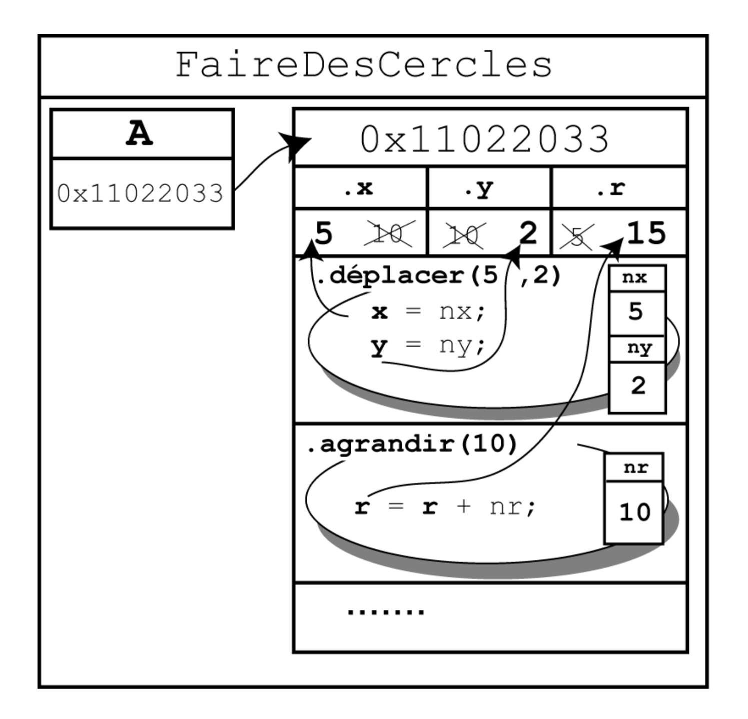


Figure 7–7.
Les méthodes appliquées à un objet exploitent les données relatives à cet objet.

```
Apres agrandissement :
Cercle centre en 5, 2
de rayon : 15
```

L'instruction `A.agrandir(10)` passe en paramètre la valeur d'accroissement du rayon du cercle. La donnée `r` de l'objet `A` est augmentée de cette valeur (*voir Figure 7-7*).

À chaque appel de la méthode `afficher()` appliquée à l'objet `A`, les valeurs courantes des données (`x`, `y` et `r`) de l'objet `A` sont affichées.

Observons que, lorsque l'objet `A` est déplacé, les deux coordonnées `x` et `y` de son centre sont modifiées. La méthode `déplacer()` modifie le contenu des deux variables d'instance `x` et `y` de l'objet `A`. Cette transformation est visible en dehors de l'objet lui-même, puisque la méthode `afficher()` affiche à l'écran le résultat de cette modification.

Résumé

La classe `String` est une classe prédéfinie du langage Java, qui définit des « variables » contenant des suites de caractères (des mots ou des chaînes de caractères).

La classe `String` est un type de données composé de méthodes, qui permettent la recherche de mots ou de caractères dans un texte. Les mots peuvent aussi être comparés suivant l'ordre alphabétique ou transformés en d'autres formats.

L'étude des objets de type `String` montre qu'une classe est une association de données (information ou valeur de tout type) et de méthodes (outils d'accès et de transformation des données). Définies dans une classe, ces méthodes ne peuvent s'appliquer qu'aux données de cette même classe.

Le langage Java offre la possibilité au programmeur de développer ses propres classes. Construire une classe, c'est définir un nouveau type. Pour cela, il est nécessaire de procéder de la façon suivante :

- Déterminer les caractéristiques communes à ce que l'on souhaite décrire. Ce sont les données, les attributs, les propriétés ou encore les membres de la classe.
- Définir toutes les opérations et traitements réalisables sur ces éléments. Ces opérations sont aussi appelées méthodes, ou encore comportements.

Une classe définissant un type structuré n'est pas une application directement exécutable. Elle ne contient pas de fonction `main()`.

Les types structurés sont utilisés dans les applications en déclarant des « variables », dont le type correspond au nom de la classe définie précédemment, comme le montre l'instruction suivante :

```
TypeDeL'Objet chose = new TypeDeL'Objet();
```

L'opérateur `new` détermine l'adresse où stocker les informations relatives à la variable déclarée. Il réserve l'espace mémoire nécessaire pour stocker les données et les méthodes de la classe. Les données sont initialisées à `0` pour les entiers, à `0.0` pour les réels, à `'\0'` pour les caractères et à `null` pour tous les autres types structurés.

À cette étape, la variable est appelée un objet dans le jargon informatique. Un objet est donc un élément particulier, qui représente une classe définissant un type structuré. On dit aussi que c'est une instance de la classe. Les données (propriétés ou attributs) qui la définissent sont appelées variables d'instance.

L'accès aux variables d'instance ainsi qu'aux méthodes de la classe se fait par l'intermédiaire de l'opérateur point (`.`), comme le montre l'exemple suivant :

```
chose.nomDeLaDonnée = valeur du bon type ;
chose.nomDeLaMéthode(liste des paramètres éventuels) ;
```

en supposant que la donnée et la méthode aient été préalablement définies pour le type de l'objet `chose`.

Exercices

Utiliser les objets de la classe `String`

7.1 Écrivez un programme qui réalise les opérations suivantes :

a. Demander la saisie d'une phrase.

b. Afficher la phrase en majuscules.

c. Compter le nombre de « `a` » dans la phrase puis, s'il y en a, transformer tous les « `a` » en « `*` ».

d. Tester si, entre le cinquième caractère et le douzième, se trouve une séquence de caractères préalablement saisie au clavier.

7.2 Écrivez un programme qui permet d'obtenir les actions suivantes :

a. Saisir des mots jusqu'à ce que l'utilisateur entre le mot « `Fin` ».

b. Afficher, parmi les mot saisis, le premier dans l'ordre alphabétique.

c. Afficher, parmi les mot saisis, le dernier dans l'ordre alphabétique.

✔ Le mot `"Fin"` ne doit pas être pris en compte dans la liste des mots saisis.

Créer une classe d'objets

7.3 L'objectif est de définir une représentation d'un objet `Personne`.

a. Sachant qu'une personne est définie à partir de son nom, son prénom et son âge, définissez les données de la classe `Personne`.

b. Écrivez une application `MesAmis` qui utilise un objet `Untel` de type `Personne` et qui demande la saisie au clavier de ses nom, prénom et âge.

Consulter les variables d'instance

7.4 Pour définir les comportements d'un objet de type `Personne` :

a. Dans la classe `Personne`, décrivez la méthode `présentezVous()`, qui affiche les caractéristiques de la personne concernée.

b. Modifiez l'application de façon à afficher les caractéristiques de l'objet `Untel`.

c. Dans la classe `Personne`, décrivez la méthode `quelEstVotreNom()`, qui permet de connaître le nom de la personne concernée.

d. Dans la classe `Personne`, décrivez la méthode `quelEstVotreAge()`, qui permet de connaître l'âge de la personne concernée.

e. Modifiez l'application de façon à afficher le nom puis l'âge d'`Untel`.

Analyser les résultats d'une application objet

7.5 Pour bien comprendre ce que réalise l'application `FaireDesPoints`, observez les deux programmes suivants :

```
public class Point // Le fichier s'appelle Point.java
{
  int x, y;

  public void créer()
  {
   System.out.print("Entrez l'abscisse : ");
   x = Lire.i();
   System.out.print("Entrez l'ordonnee : ");
   y = Lire.i();
  }
  public void afficher()
  {
   System.out.println("x : " + x + " y : " + y);
  }
  public void déplacer( int nx, int ny)
  {
   x = nx;
   y = ny;
  }
} // fin de la class Point
```

```
public class FaireDesPoints // Le fichier s'appelle FaireDesPoints.java
{
 public static void main( String [] arg)
 {
  Point P = new Point();
  P.afficher();
  P.créer();
  P. afficher();
  P.déplacer(10, 12);
  P.afficher();
 }
   } // fin de la class FaireDesPoints
```

a. Quel est le programme qui correspond à l'application ?

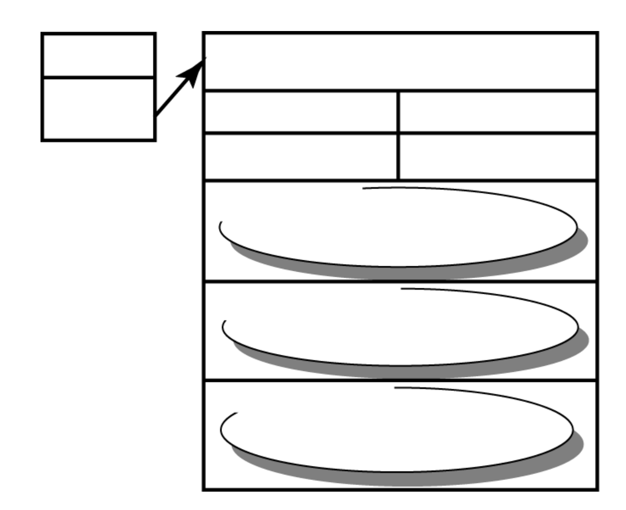
b. Quel est le programme définissant le type `Point` ?

c. Recherchez les attributs de la classe `Point`, et donnez leur nom.

d. Combien de méthodes sont-elles définies dans la classe `Point` ? Donnez leur nom.

e. Quels sont les objets utilisés par l'application `FaireDesPoints` ? Que valent leurs données `x` et `y` après exécution de l'instruction déclaration ?

f. Sur la représentation graphique ci-dessous, placez, pour l'objet `P`, la valeur initiale ainsi que le nom des méthodes.



g. À l'appel de la méthode `créer()`, comment les valeurs sont-elles affectées aux attributs des objets concernés ? Modifiez les cases concernées sur la représentation graphique.

h. Même question pour la méthode `déplacer()`.

i. Quel est le résultat final de l'application ?

Le projet « Gestion d'un compte bancaire »

Traiter les chaînes de caractères

Le `type` d'un compte et son `numéro` ne sont plus définis respectivement comme `char` et `long` mais comme deux objets de type `String`. Le type d'un compte peut donc prendre maintenant les thèmes `courant`, `joint` ou `épargne`.

a. Saisissez le `type` du compte de façon que l'utilisateur entre au clavier `C`, `J` ou `E`. Le programme place dans la variable `type` les chaînes `courant`, `joint` ou `épargne` en fonction de la lettre saisie.

b. Saisissez le `numéro` de compte sous la forme d'une chaîne de caractères.

c. Transformez tous les tests faisant appel aux variables `type` et `numéro` de façon à tester non plus sur des caractères mais sur des `String`.

Définir le type `Compte`

Dans un fichier nommé `Compte.java`, définissez la classe `Compte` en procédant de la façon suivante :

a. Déterminez les données qui définissent tout compte bancaire.

b. Écrivez les méthodes associées, par exemple :

– `créerCpte()`, en reprenant les instructions de l'option `1`, décrites au chapitre précédent. Placez-les sous l'en-tête de la fonction qui a pour forme `public void`

`créerCpte()`. La méthode ne possède ni paramètre, ni type de retour, car elle ne fait que modifier les données caractéristiques d'un compte déclaré en dehors de la méthode.

– `afficherCpte()`, en reprenant la fonction écrite au chapitre précédent et en supprimant le mot-clé `static`. Les variables `num`, `type`, `taux` et `val` ne sont à déclarer ni à l'intérieur, ni en paramètre de la méthode. Elles sont définies comme données de la classe `compte`, en dehors de la méthode.

Construire l'application `Projet`

Dans un fichier nommé `Projet.java`, écrivez l'application contenant la fonction `main()` en procédant de la façon suivante :

a. Faites appel aux fonctions `alAide()`, `sortir()` et `menuPrincipal()`.

b. Créez un objet de type `Compte` grâce à l'instruction de déclaration `Compte c = new Compte();`.

c. Dans les options appropriées du menu, appelez les méthodes de la classe `Compte`, comme `c.créerCpte()` ou `c.afficherCpte()`.

d. À l'exécution du programme, remarquez que la méthode `afficherCpte()` affiche les différentes valeurs du compte, modifiées par la méthode `créerCpte()`. Une méthode par l'intermédiaire d'un objet peut, par conséquent, transmettre plusieurs résultats.

Définir le type `LigneComptable`

Dans un fichier nommé `LigneComptable.java`, définissez la classe `LigneComptable` en procédant de la façon suivante :

a. Déterminez les données qui définissent tout compte bancaire.

✔ Voir, au chapitre introductif, *« Naissance d'un programme »*, la définition de l'option 3, à la section *« Le projet "Gestion d'un compte bancaire" »*.

b. Écrivez les méthodes associées, par exemple :

– `créerLigneComptable()`, qui demande la saisie au clavier des valeurs correspondant aux données de la classe LigneComptable.

– `afficherLigne()`, qui affiche les données caractéristiques d'une ligne comptable.

Modifier le type `Compte`

Dans le fichier `Compte.java` :

a. Définissez une nouvelle donnée (variable d'instance) décrivant une ligne comptable, en écrivant la déclaration `LigneComptable ligne ;` au même niveau que `type`, `numéro`, etc.

b. Écrivez la méthode `créerLigne()` qui permet les actions suivantes :

– créer en mémoire l'objet `ligne` grâce à l'instruction de déclaration `ligne = new LigneComptable() ;`

- faire appel à la méthode `créerLigneComptable()` par l'intermédiaire de l'objet `ligne` de façon à enregistrer les valeurs numériques associées à la ligne créée ;
- modifier la valeur courante du compte à partir de la valeur (débit ou crédit) saisie dans la méthode `créerLigneComptable()`.

c. Modifiez la méthode `afficherCpte()` de façon à afficher les informations stockées dans `ligne`, en utilisant l'instruction `ligne.afficherLigne()` .

Modifier l'application `Projet`

a. Dans le fichier nommé `Projet.java`, modifiez l'option `3` de l'application de façon qu'une ligne comptable soit créée pour le compte `C`, défini à l'option `1`.

b. À l'exécution de l'application, que se passe-t-il si l'utilisateur ayant créé un compte affiche ce dernier sans avoir jamais créé de ligne comptable ? Pourquoi ?

c. Comment faire pour remédier à cette situation ?

8

Les principes du concept d'objet

Au cours du chapitre précédent, nous avons examiné comment mettre en place des objets à l'intérieur d'un programme Java. Cette étude a montré combien la structure générale des programmes se trouvait modifiée par l'emploi des objets.

En réalité, les objets sont beaucoup plus qu'une structure syntaxique. Ils sont régis par des principes essentiels, qui constituent les fondements de la programmation objet. Dans ce chapitre, nous étudions avec précision l'ensemble de ces principes.

Nous déterminons d'abord (*section « La communication objet »*) les caractéristiques d'une donnée `static` et évaluons leurs conséquences sur la construction des objets en mémoire. Nous analysons également la technique du passage de paramètres par référence. Nous observons qu'il est possible, avec la technologie objet, qu'une méthode transmette plusieurs résultats à une autre méthode.

Nous expliquons ensuite (*section « Les objets contrôlent leur fonctionnement »*), le concept d'encapsulation des données, et nous examinons pourquoi et comment les objets protègent leurs données.

Enfin, nous définissons (*section « L'héritage »*) la notion d'héritage entre classes. Nous observons combien cette notion est utile puisqu'elle permet de réutiliser des programmes tout en apportant des variations dans le comportement des objets héritants.

La communication objet

En définissant un type ou une classe, le développeur crée un modèle, qui décrit les fonctionnalités des objets utilisés par le programme. Les objets sont créés en mémoire à partir de ce modèle, par copie des données et des méthodes.

Cette copie est réalisée lors de la réservation des emplacements mémoire grâce à l'opérateur `new`, qui initialise les données de l'objet et fournit, en retour, l'adresse où se trouvent les informations stockées.

La question est de comprendre pourquoi l'interpréteur réalise cette copie en mémoire, alors que cela lui était impossible auparavant.

Les données `static`

La réponse à cette interrogation se trouve dans l'observation des différents programmes proposés dans ce manuel (*voir les chapitres 6, « Fonctions, notions avancées », et 7, « Les classes et les objets »*). Comme nous l'avons déjà constaté (*voir, au chapitre précédent, la section « Construire et utiliser ses propres classes »*), le mot-clé `static` n'est plus utilisé lors de la description d'un type, alors qu'il était présent dans tous les programmes précédant ce chapitre.

C'est donc la présence ou l'absence de ce mot-clé qui fait que l'interpréteur construise ou non des objets en mémoire.

Lorsque l'interpréteur rencontre le mot-clé `static` devant une variable ou une méthode, il réserve un seul et unique emplacement mémoire pour y stoker la valeur ou le pseudo-code associés. Cet espace mémoire est communément accessible pour tous les objets du même type.

Lorsque le mot-clé `static` n'apparaît pas, l'interpréteur réserve, à chaque appel de l'opérateur `new`, un espace mémoire pour y charger les données et les pseudo-codes décrits dans la classe.

Exemple : compter des cercles

Pour bien comprendre la différence entre une donnée `static` et une donnée non `static`, nous allons modifier la classe `Cercle`, de façon qu'il soit possible de connaître le nombre d'objets `Cercle` créés en cours d'application.

Pour ce faire, l'idée est d'écrire une méthode `créer()`, qui permette, d'une part, de saisir des valeurs `x`, `y` et `r` pour chaque cercle à créer et, d'autre part, d'incrémenter un compteur de cercles.

La variable représentant ce compteur doit être indépendante des objets créés, de sorte que sa valeur ne soit pas être réinitialisée à zéro à chaque création d'objet. Cette variable doit cependant être accessible pour chaque objet de façon qu'elle puisse s'incrémenter de `1` à chaque appel de la méthode `créer()`.

Pour réaliser ces contraintes, le compteur de cercles doit être une variable de classe, c'est-à-dire une variable déclarée avec le mot-clé `static`. Examinons tout cela dans le programme suivant.

```
public class Cercle  {
 public int x, y, r ; // position du centre et rayon
 public static int nombre; // nombre de cercle

  public void créer()  {
   System.out.print(" Position en x :  ");
   x = Lire.i();
   System.out.print(" Position en y :  ");
   y = Lire.i();
   System.out.print(" Rayon           :  ");
```

```
    r = Lire.i();
    nombre ++;
   }
   // et toutes les autres méthodes de la classe Cercle définies au
   // chapitre précédent
 } // Fin de la classe Cercle
```

Les données définies dans la classe `Cercle` sont de deux sortes : les variables d'instance, `x`, `y` et `r`, et la variable de classe, `nombre`. Seul le mot-clé `static` permet de différencier leur catégorie.

Grâce au mot-clé `static`, la variable de classe `nombre` est un espace mémoire commun, accessible pour tous les objets créés. Pour faire appel à cette variable, il suffit de l'appeler par son nom véritable (*voir, au chapitre 6, « Fonctions, notions avancées », la section « Variable de classe »*), c'est-à-dire `nombre`, si elle est utilisée dans la classe `Cercle`, ou `Cercle.nombre`, si elle est utilisée en dehors de cette classe.

Exécution de l'application `CompterDesCercles`

Pour mieux saisir la différence entre les variables d'instance (non `static`) et les variables de classe (`static`), observons comment fonctionne l'application `CompterDesCercles`.

```
public class CompterDesCercles  {
  public static void main(String [] arg)
  {
   Cercle A  = new Cercle();
   A.créer();
   System.out.println("Nombre de cercle : " + Cercle.nombre);

   Cercle B = new Cercle();
   B.créer();
   System.out.println("Nombre de cercle : " + Cercle.nombre);
  }
} // Fin de la classe CompterDesCercles
```

Dans ce programme, deux objets de type `Cercle` sont créés à partir du modèle défini par le type `Cercle`. Chaque objet est un représentant particulier, ou une instance, de la classe `Cercle`, de position et de rayon spécifiques.

Lorsque l'objet `A` est créé en mémoire, grâce à l'opérateur `new`, les données `x`, `y` et `r` sont initialisées à `0` au moment de la réservation de l'espace mémoire. La variable de classe `nombre` est elle aussi créée en mémoire, et sa valeur est également initialisée à `0`.

Lors de l'exécution de l'instruction `A.créer();`, les valeurs des variables `x`, `y` et `r` de l'instance `A` sont saisies au clavier (`x = Lire.i(), …`). La variable de classe `nombre` est incrémentée de `1` (`nombre++`). Le nombre de cercles est alors de `1` (*voir l'objet* `A`*, décrit à la Figure 8-1*).

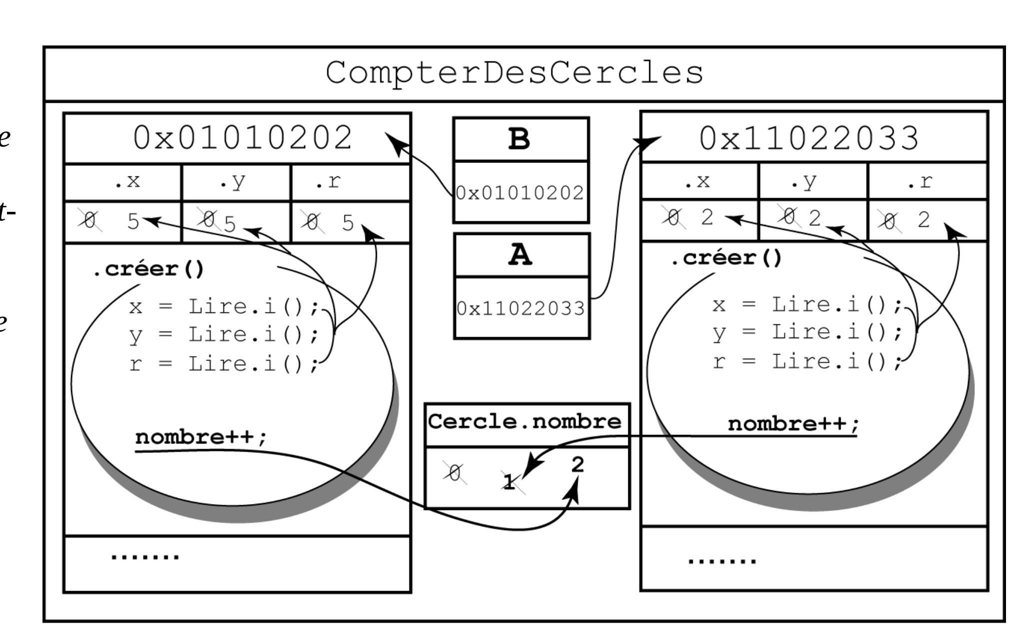


Figure 8-1.
La variable de classe `Cercle.nombre` *est créée en mémoire, avec l'objet* `A`*. Grâce au mot-clé* `static`*, il y a, non pas réservation d'un nouvel espace mémoire (pour la variable* `nombre`*) lors de la création de l'objet* `B`*, mais préservation de l'espace mémoire ainsi que de la valeur stockée.*

De la même façon, l'objet `B` est créé en mémoire grâce à l'opérateur `new`. Les données `x`, `y` et `r` sont, elles aussi, initialisées à `0`.

Pour la variable de classe `nombre`, en revanche, cette initialisation n'est pas réalisée. La présence du mot-clé `static` fait que la variable de classe `nombre`, qui existe déjà en mémoire, ne peut être réinitialisée directement par l'interpréteur.

Il y a donc, non pas réservation d'un nouvel emplacement mémoire, mais préservation du même emplacement mémoire, avec conservation de la valeur calculée à l'étape précédente, soit `1`.

Après saisie des données `x`, `y` et `r` de l'instance `B`, l'instruction `nombre++` fait passer la valeur de `Cercle.nombre` à `2` (*voir l'objet* `B` *décrit à la Figure 8-1*).

N'existant qu'en un seul exemplaire, la variable de classe `nombre` permet le comptage du nombre de cercles créés. L'incrémentation de cette valeur est réalisée indépendamment de l'objet, la variable étant commune à tous les objets créés.

Le passage de paramètres par référence

La communication des données entre les objets passe avant tout par l'intermédiaire des variables d'instance. Nous l'avons observé à la section précédente, lorsqu'une méthode appliquée à un objet modifie les valeurs de plusieurs données de cet objet, cette modification est visible en dehors de la méthode et de l'objet lui-même.

Il existe cependant une autre technique qui permette la modification des données d'un objet : le passage de **paramètres par référence**.

Ce procédé est utilisé lorsqu'on passe en paramètre d'une méthode, non plus une simple variable (de type `int`, `char` ou `double`), mais un objet. Dans cette situation, l'objet étant défini par son adresse (référence), la valeur passée en paramètre n'est plus la valeur réelle de la variable mais l'adresse de l'objet.

Grâce à cela, les modifications apportées sur l'objet passé en paramètre et réalisées à l'intérieur de la méthode sont visibles en dehors même de la méthode.

Échanger la position de deux cercles

Pour comprendre en pratique le mécanisme du passage de paramètres par référence, nous allons écrire une application qui échange la position des centres de deux cercles donnés.

Pour cela, nous utilisons le mécanisme d'échange de valeurs (*voir le chapitre 1, « Stocker une information »*), en l'appliquant à la coordonnée `x` puis à la coordonnée `y` des centres des deux cercles à échanger.

Examinons la méthode `échanger()`, dont le code ci-dessous s'insère dans la classe `Cercle`.

✔ Voir, au chapitre 7, *« Les classes et les objets »*, la section *« La classe descriptive du type* `Cercle` *»*.

```
public void échanger(Cercle autre)    {        // Échange la position d'un
  int tmp;                          // cercle avec celle du cercle donné en paramètre
  tmp = x;                          // échanger la position en x
  x = autre.x;
  autre.x = tmp;
  tmp = y;                          // échanger la position en y
  y = autre.y;
  autre.y = tmp;
}
```

Pour échanger les coordonnées des centres de deux cercles, la méthode `échanger()` doit avoir accès aux valeurs des coordonnées des deux centres des cercles concernés.

Si, par exemple, la méthode est appliquée au cercle `B` (`B.échanger()`), ce sont les variables d'instance `x` et `y` de l'objet `B` qui sont modifiées par les coordonnées du centre du cercle `A`. La méthode doit donc connaître les coordonnées du cercle `A`. Pour ce faire, il est nécessaire de passer ces valeurs en paramètres de la fonction.

La technique consiste à passer en paramètres, non pas les valeurs `x` et `y` du cercle avec lequel l'échange est réalisé, mais un objet de type `Cercle`. Dans notre exemple, ce paramètre s'appelle `autre`. C'est le paramètre formel de la méthode représentant n'importe quel cercle, et il peut donc représenter, par exemple, le cercle `A`.

Le fait d'échanger les coordonnées des centres de deux cercles revient à échanger les coordonnées du couple (`x`, `y`) du cercle sur lequel on applique la méthode (`B.x`, `B.y`) avec les coordonnées (`autre.x`, `autre.y`) du cercle passé en paramètre de la méthode (`A.x`, `A.y`).

Examinons maintenant comment s'opère effectivement l'échange en exécutant l'application suivante :

```
public class EchangerDesCercles    {
 public static void main(String [] arg)    {
  Cercle A  = new Cercle();
  A.créer();
  System.out.println("Le cercle A : ");
  A.afficher();

  Cercle B = new Cercle();
  B.créer();
  System.out.println("Le cercle B : ");
```

```
    B. afficher() ;

    B.échanger(A) ;
    System.out.println("Apres echange, ") ;
    System.out.println("Le cercle A : ") ;
    A.afficher() ;
    System.out.println("Le cercle B : ") ;
    B.afficher() ;
  }
}
```

Exécution de l'application `EchangerDesCercles`

Nous supposons que l'utilisateur ait saisi les valeurs suivantes, pour le cercle `A`

```
Position en x :  2
Position en y :  2
Rayon         :  2
Le cercle A :
Centre en 2, 2
Rayon : 2
```

et pour le cercle `B`

```
Position en x :  5
Position en y :  5
Rayon         :  5
Le cercle B :
Centre en 5, 5
Rayon : 5
```

L'instruction `B.échanger(A)` échange les coordonnées (`x`, `y`) de l'objet `B` avec celles de l'objet `A`. C'est donc le pseudo-codede l'objet `B` qui est interprété, comme illustré à la Figure 8-2.

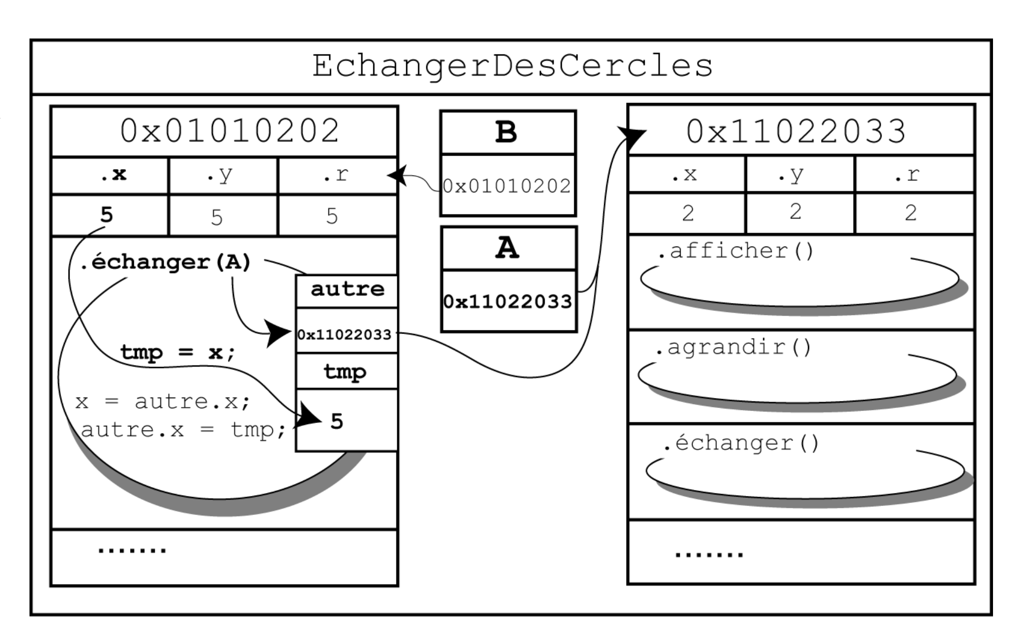


Figure 8-2.

L'instruction `B.échanger(A)` *fait appel à la méthode* `échanger()` *de l'objet* `B`*. Les données* `x`, `y` *et* `r` *utilisées par cette méthode sont celles de l'objet* `B`*.*

Examinons le tableau d'évolution des variables déclarées pour le pseudo-code de l'objet `B`.

instruction	tmp	x	y	autre
`valeurs initiales`	-	5	5	0x11022033

- À l'entrée de la méthode, la variable `tmp` est déclarée sans être initialisée.
- La méthode est appliquée à l'objet `B`. Les variables `x` et `y` de l'instance `B` ont pour valeurs respectives `5` et `5`.
- L'objet `autre` est simplement déclaré en paramètre de la fonction `échanger(Cercle autre)`. L'opérateur `new` n'étant pas appliqué à cet objet, aucun espace mémoire supplémentaire n'est alloué.

  Comme `autre` représente un objet de type `Cercle`, il ne peut contenir qu'une adresse et non pas une simple valeur numérique. Cette adresse est celle du paramètre effectivement passé lors de l'appel de la méthode.

  Pour notre exemple, l'objet `A` est passé en paramètre de la méthode `(B.échanger(A))`. La case mémoire de la variable `autre` prend donc pour valeur l'adresse de l'objet `A`.

instruction	tmp	x	autre	autre.x (A.x)
`tmp = x ;`	5	5	0x11022033	2
`x = autre.x ;`	5	2	0x11022033	2
`autre.x = tmp ;`	5	2	0x11022033	5

- La variable `tmp` prend ensuite la valeur de la coordonnée `x` de l'objet `B`, soit `5`.

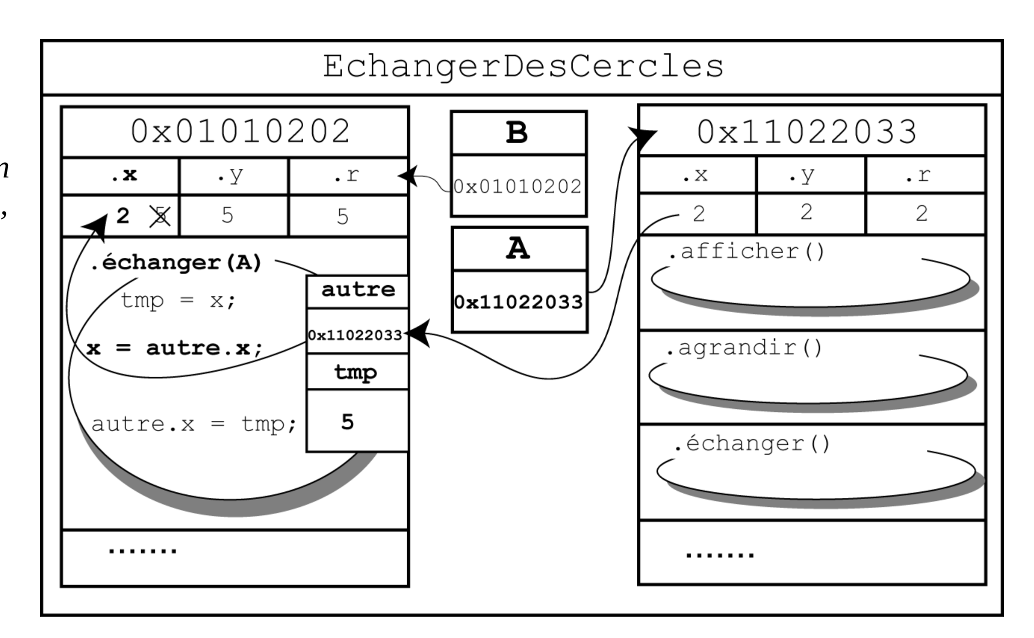


Figure 8-3.

L'objet `autre` *est le paramètre formel de la méthode* `échanger()`*. En écrivant* `B.échanger(A)`*, l'objet* `autre` *stocke la référence mémorisée en* `A`*. De cette façon,* `autre.x` *représente également* `A.x`*. La variable* `x` *de l'instance* `B` *prend la valeur de* `A.x` *grâce à l'instruction* `x = autre.x`*.*

- Lorsque l'instruction `x = autre.x` est exécutée, la coordonnée `x` de l'objet `B` prend la valeur de la coordonnée `x` de l'objet `autre.x`. Puisque `autre` correspond à l'adresse de l'objet `A`, le fait de consulter le contenu de `autre.x` revient, en réalité, à consulter le contenu de `A.x` (*voir Figure 8-3*). La variable d'instance `A.x` contenant la valeur `2`, `x` (`B.x`) prend la valeur `2`.
- Pour finir, l'échange sur les abscisses, `autre.x`, prend la valeur stockée dans `tmp`. Comme `autre` et `A` correspondent à la même adresse, modifier `autre.x`, c'est aussi modifier `A.x` (*voir Figure 8-4*). Une fois exécuté `autre.x = tmp`, la variable `x` de l'instance `A` vaut par conséquent `5`.

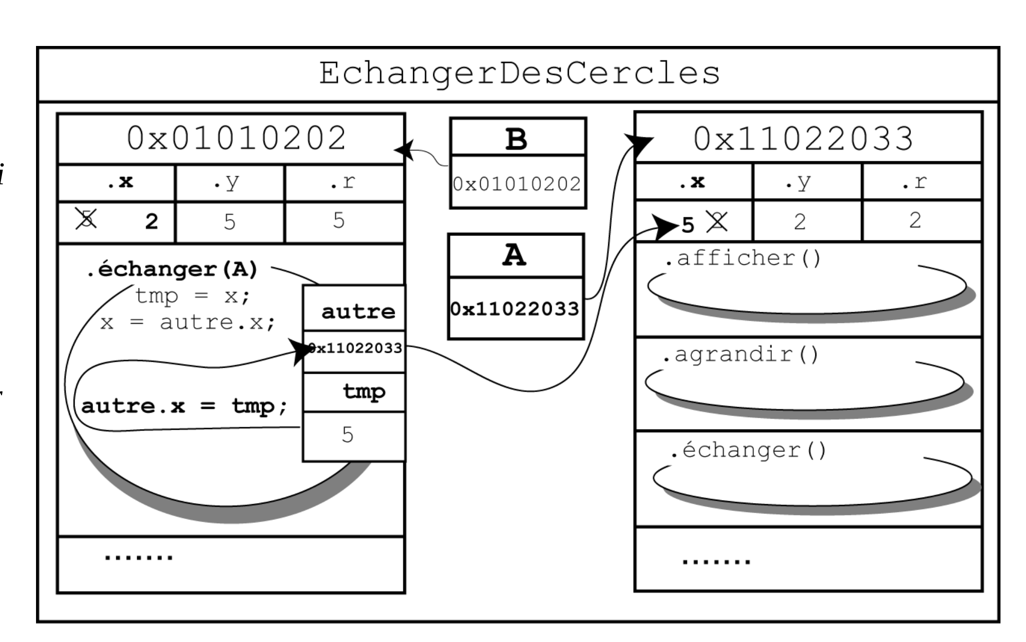


Figure 8-4.
`autre` et `A` définissent la même référence, ou adresse. C'est pourquoi le fait de modifier `autre.x` revient aussi à modifier `A.x`. Ainsi, l'instruction `autre.x = tmp` fait que `A.x` prend la valeur stockée dans `tmp`.

L'ensemble de ces opérations est ensuite réalisé sur la coordonnée `y` des cercles `B` et `A` *via* `autre`.

instruction	tmp	y	autre	autre.y (A.y)
`tmp = y ;`	5	5	0x11022033	2
`y = autre.y ;`	5	2	0x11022033	2
`autre.y = tmp ;`	5	2	0x11022033	5

L'exécution finale du programme a pour résultat

```
Apres echange,
Le cercle A :
Centré en 5, 5
Rayon : 2
Le cercle B :
Centre en 2, 2
Rayon : 5
```

Au final, nous constatons, à l'observation des tableaux d'évolution des variables, que les données `x` et `y` de `B` ont pris la valeur des données `x` et `y` de `A`, soit `2` pour `x` et `2` pour `y`. Parallèlement, le cercle `A` a été transformé par l'intermédiaire de la référence stockée dans `autre` et a pris les coordonnées `x` et `y` du cercle `B`, soit `5` pour `x` et `5` pour `y`.

En résumé, grâce à la technique du passage de paramètres par référence, tout objet passé en paramètre d'une méthode voit, en sortie de la méthode, ses données transformées par la méthode. Cette transformation est alors visible pour tous les objets de l'application.

Les objets contrôlent leur fonctionnement

L'un des objectifs de la programmation objet est de simuler, à l'aide d'un programme informatique, la manipulation des objets réels par l'être humain. Les objets réels forment un tout, et leur manipulation nécessite la plupart du temps un outil, ou une interface, de communication.

Par exemple, quand nous prenons un ascenseur, nous appuyons sur le bouton d'appel pour ouvrir les portes ou pour nous rendre jusqu'à l'étage désiré. L'interface de communication est ici le bouton d'appel. Nul n'aurait l'idée de prendre la télécommande de sa télévision pour appeler un ascenseur.

De la même façon, la préparation d'une omelette nécessite de casser des œufs. Pour briser la coquille d'un œuf, nous pouvons utiliser l'outil couteau. Un marteau pourrait être également utilisé, mais son usage n'est pas vraiment adapté à la situation.

Comme nous le constatons à travers ces exemples, les objets réels sont manipulés par l'intermédiaire d'interfaces **appropriées**. L'utilisation d'un outil inadapté fait que l'objet ne répond pas à nos attentes ou qu'il se brise définitivement.

Tout comme nous manipulons les objets réels, les applications informatiques manipulent des objets virtuels, définis par le programmeur. Cette manipulation nécessite des outils aussi bien adaptés que nos outils réels. Sans contrôle sur le bien-fondé d'une manipulation, l'application risque de fournir de mauvais résultats ou, pire, de cesser brutalement son exécution.

La notion d'encapsulation

Pour réaliser l'adéquation entre un outil et la manipulation d'un objet, la programmation objet utilise le concept d'**encapsulation**.

Par ce terme, il faut entendre que les données d'un objet sont protégées, tout comme le médicament est protégé par la fine pellicule de sa capsule. Grâce à cette protection, il ne peut y avoir transformation involontaire des données de l'objet.

L'encapsulation passe par le contrôle des données et des comportements de l'objet. Ce contrôle est établi à travers la protection des données (*voir la section suivante*), l'accès contrôlé aux données (*voir la section « Les méthodes d'accès aux données »*) et la notion de constructeur de classe (*voir la section « Les constructeurs »*).

La protection des données

Le langage Java fournit les niveaux de protection suivants pour les membres d'une classe (données et méthodes) :

- **Protection** `public`**.** Les membres (données et méthodes) d'une classe déclarés `public` sont accessibles pour tous les objets de l'application. Les données peuvent être modifiées par une méthode de la classe, d'une autre classe ou depuis la fonction `main()`.
- **Protection** `private`**.** Les membres de la classe déclarés `private` ne sont accessibles que pour les méthodes de la même classe. Les données ne peuvent être initialisées ou modifiées que par l'intermédiaire d'une méthode de la classe. Les données ou méthodes ne peuvent être appelées par la fonction `main()`.
- **Protection** `protected`**.** Tout comme les membres privés, les membres déclarés `protected` ne sont accessibles que pour les méthodes de la même classe. Ils sont aussi accessibles par les fonctions membres d'une sous-classe (*voir la section « L'héritage »*).

Par défaut, lorsque les données sont déclarées sans type de protection, leur protection est `public`. Les données sont alors accessibles depuis toute l'application.

Protéger les données d'un cercle

Pour protéger les données de la classe `Cercle`, il suffit de remplacer le mot-clé `public` précédant la déclaration des variables d'instance par le mot `private`. Observons la nouvelle classe, `CerclePrive`, dont les données sont ainsi protégées.

```
public class CerclePrive
{
 private int x, y, r ; // position du centre et rayon

 public void afficher() {
  // voir la section "La classe descriptive du type Cercle" du chapitre
  //"Les classes et les objets"
    }

 public double périmètre() {
  // voir la section "La classe descriptive du type Cercle" du chapitre
  //"Les classes et les objets"
    }
 public void déplacer(int nx, int ny) {
  // voir la section "La classe descriptive du type Cercle" du chapitre
  //"Les classes et les objets"
    }

 public void agrandir(int nr) {
  // voir la section "La classe descriptive du type Cercle" du chapitre
  //"Les classes et les objets"
    }
} // Fin de la classe CerclePrive
```

Les données `x`, `y` et `r` de la classe `CerclePrive` sont protégées grâce au mot-clé `private`. Étudions les conséquences d'une telle protection sur la phase de compilation de l'application `FaireDesCerclesPrives`.

```
public class FaireDesCerclesPrives
{
 public static void main(String [] arg)
 {
  CerclePrive A  = new CerclePrive();
  A.afficher();
  System.out.println(" Entrez le rayon : ");
  A.r = Lire.i() ;
  System.out.println(" Le cercle est de rayon : " + A.r) ;
 }
}
```

Compilation de l'application `FaireDesCerclesPrives`

Les données `x`, `y` et `r` de la classe `CerclePrive` sont déclarées privées. Par définition, ces données ne sont donc pas accessibles en dehors de la classe où elles sont définies.

Or, en écrivant dans la fonction `main()` l'instruction `A.r = Lire.i() ;`, le programmeur demande d'accéder, depuis la classe `FaireDesCerclesPrives`, à la valeur de `r`, de façon à la modifier. Cet accès est impossible, car `r` est défini en mode `private` dans la classe `CerclePrive` et non dans la classe `FaireDesCerclesPrives`.

C'est pourquoi le compilateur détecte l'erreur `Variable x in class CerclePrive not accessible from class FaireDesCerclesPrives`.

Les méthodes d'accès aux données

Lorsque les données sont totalement protégées, c'est-à-dire déclarées `private` à l'intérieur d'une classe, elles ne sont plus accessibles depuis une autre classe ou depuis la fonction `main()`. Pour connaître ou modifier la valeur d'une donnée, il est nécessaire de créer, à l'intérieur de la classe, des méthodes d'accès à ces données.

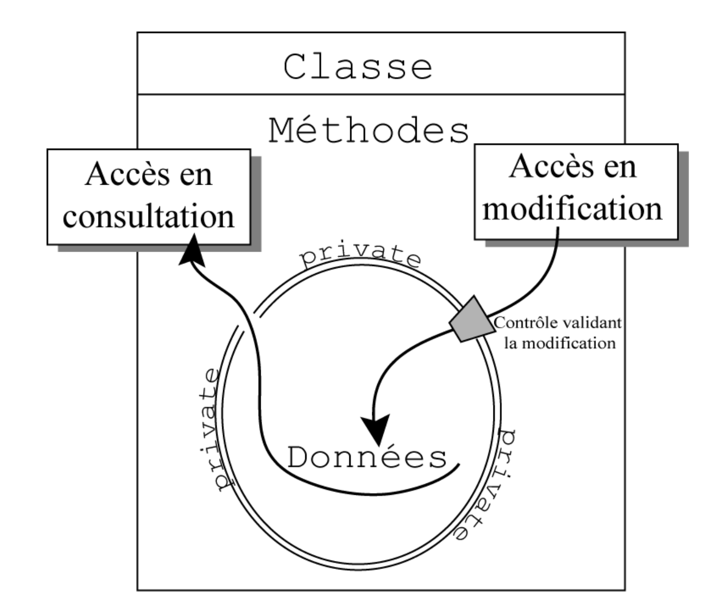


Figure 8-5.

Lorsque les données d'un objet sont protégées, l'objet possède ses propres méthodes, qui permettent soit de consulter la valeur réelle de ses données, soit de modifier les données. La validité de ces modifications est contrôlée par les méthodes définies dans la classe.

Les données privées ne peuvent être consultées ou modifiées que par des méthodes de la classe où elles sont déclarées.

De cette façon, grâce à l'accès aux données par l'intermédiaire de méthodes appropriées, l'objet permet, non seulement la consultation de la valeur de ses données, mais aussi l'autorisation ou non, suivant ses propres critères, de leur modification.

Les méthodes d'une classe réalisent les modes d'accès suivants :

- **Accès en consultation.** La méthode fournit la valeur de la donnée mais ne peut la modifier. Ce type de méthode est aussi appelé **accesseur** en consultation.
- **Accès en modification.** La méthode modifie la valeur de la donnée. Cette modification est réalisée après validation par la méthode. On parle aussi d'accesseur en modification.

Contrôler les données d'un cercle

Dans l'exemple suivant, nous prenons pour hypothèse que le rayon d'un cercle ne puisse jamais être négatif ni dépasser la taille de l'écran. Ces conditions doivent être vérifiées pour toutes les méthodes qui peuvent modifier la valeur du rayon d'un cercle.

Comme nous l'avons déjà remarqué (*voir, au chapitre 7, « Les classes et les objets », la section « Quelques observations »*), les méthodes `afficher()` et `périmètre()` ne font que consulter le contenu des données `x`, `y` et `r`.

Les méthodes `déplacer()`, `agrandir()` et `créer()`, en revanche, modifient le contenu des données `x`, `y` et `r`. La méthode `déplacer()` n'ayant pas d'influence sur la donnée `r`, seules les méthodes `agrandir()` et `créer()` doivent contrôler la valeur du rayon, de sorte que cette dernière ne puisse être négative ou supérieure à la taille de l'écran. Examinons la classe `CercleControle` suivante, qui prend en compte ces nouvelles contraintes :

```
public class CercleControle    {
 private int x, y, r ; // position du centre et rayon
  public void créer()      {
   System.out.print(" Position en x :  ");
   x = Lire.i();
   System.out.print(" Position en y :  ");
   y = Lire.i();
   do  {
     System.out.print(" Rayon            :  ");
     r = Lire.i();
   } while ( r < 0 || r >  600);
  }

 public void afficher()     {    //Affichage des données de la classe
  System.out.println(" Centre en " + x + ", " + y);
  System.out.println(" Rayon : " + r);
 }

 public void agrandir(int nr)     {
   if (r + nr < 0) r = 0;
     else if ( r + nr > 600) r = 600;
     else r = r + nr;
```

```
  }
} // Fin de la classe CercleControle
```

La méthode `créer()` contrôle la valeur du rayon lors de sa saisie, en demandant de saisir une valeur pour le rayon tant que la valeur saisie est négative ou plus grande que `600` (taille supposée de l'écran). Dès que la valeur saisie est comprise entre `0` et `600`, la fonction `créer()` cesse son exécution. À la sortie de cette fonction, nous sommes certains que le rayon est compris entre `0` et `600`.

De la même façon, la méthode `agrandir()` autorise que la valeur du rayon soit augmentée de la valeur passée en paramètre, à condition que cette augmentation ne dépasse pas la taille de l'écran ou que la diminution n'entraîne pas un rayon négatif, si la valeur passée en paramètre est négative. Dans ces deux cas, la valeur du rayon est forcée respectivement à la taille de l'écran ou à `0`.

Exécution de l'application `FaireDesCerclesControles`

Pour vérifier que tous les objets `Cercle` contrôlent bien la valeur de leur rayon, examinons l'exécution de l'application suivante :

```
public class FaireDesCerclesControles        {
 public static void main(String [] arg)     {
  CercleControle A  = new CercleControle();
  A.créer();
  A.afficher();
  System.out.print("Entrer une valeur d'agrandissement :");
  int plus = Lire.i();
  A.agrandir(plus);
  System.out.println("Apres agrandissement : ");
  A.afficher();
 }
}
```

L'objet `A` est créé en mémoire grâce à l'opérateur `new`. La valeur du rayon est initialisée à `0`. À l'appel de la méthode `créer()`, les variables d'instance `x` et `y` sont saisies au clavier, comme suit :

```
Position en x :  5
Position en y :  5
```

Ensuite, si l'utilisateur saisit pour le rayon une valeur négative

```
Rayon         :  -3
```

ou supérieure à 600

```
Rayon         :  654
```

le programme demande de nouveau de saisir une valeur pour le rayon. L'application cesse cette répétition lorsque l'utilisateur entre une valeur comprise entre `0` et `600`, comme suit :

```
Rayon         :  200
Centre 5, 5
Rayon : 200
```

Après affichage des données du cercle `A`, le programme demande

```
Entrer une valeur d'agrandissement : 450
```

La valeur du rayon vaut `200 + 450`, soit `650`. Ce nouveau rayon étant supérieur à `600`, la valeur du rayon est bloquée par le programme à `600`. L'affichage des données fournit

```
Apres agrandissement :
Centre 5, 5
Rayon : 600
```

La notion de constante

D'une manière générale, en programmation objet, les variables d'instance ne sont que très rarement déclarées en `public`. Pour des raisons de sécurité, tout objet se doit de contrôler les transformations opérées par l'application sur lui-même. C'est pourquoi les données d'une classe sont le plus souvent déclarées en mode `private`.

Il existe des données, appelées **constantes** qui, parce qu'elles sont importantes, doivent être visibles par toutes les méthodes de l'application. Ces données sont déclarées en mode `public`. Du fait de leur invariabilité, l'application ne peut modifier leur contenu.

Pour notre exemple, la valeur `600`, correspondant à la taille (largeur et hauteur) supposée de l'écran, peut être considérée comme une donnée constante de l'application.

Il suffit de déclarer les variables « constantes » à l'aide du mot-clé `final`. Ainsi, la taille de l'écran peut être définie de la façon suivante :

```
public final int TailleEcran = 600 ;
```

Notons que la taille de l'écran est une valeur indépendante de l'objet `Cercle`. Quelle que soit la forme à dessiner (carré, cercle, etc.), la taille de l'écran est toujours la même. C'est pourquoi il est logique de déclarer la variable `TailleEcran` comme constante de classe à l'aide du mot-clé `static`.

```
public final static int TailleEcran = 600 ;
```

De cette façon, la variable `TailleEcran` est accessible en consultation depuis toute l'application, mais elle ne peut en aucun cas être modifiée, étant déclarée `final`.

Les méthodes `créer()` et `agrandir()` s'écrivent alors de la façon suivante :

```
public void créer()    {
   System.out.print(" Position en x :  ");
   x = Lire.i();
   System.out.print(" Position en y :  ");
   y = Lire.i();
   do  {
     System.out.print(" Rayon            :  ");
     r = Lire.i();
   } while ( r < 0 || r > TailleEcran) ;
}

public void agrandir(int nr)  {
   if (r + nr < 0) r = 0;
```

```
    else if ( r + nr > TailleEcran) r = TailleEcran ;
    else r = r + nr;
  }
```

Des méthodes invisibles

Comme nous l'avons observé précédemment, les données d'une classe sont généralement déclarées en mode `private`. Les méthodes, quant à elles, sont le plus souvent déclarées `public`, car ce sont elles qui permettent l'accès aux données protégées. Dans certains cas particuliers, il peut arriver que certaines méthodes soient définies en mode `private` . Elles deviennent alors inaccessibles depuis les classes extérieures.

Ainsi, le contrôle systématique des données est toujours réalisé par l'objet lui-même, et non par l'application qui utilise les objets. Par conséquent, les méthodes qui ont pour charge de réaliser cette vérification peuvent être définies comme méthodes internes à la classe puisqu'elles ne sont jamais appelées par l'application.

Par exemple, le contrôle de la validité de la valeur du rayon n'est pas réalisée par l'application `FaireDesCercles` mais correspond à une opération interne à la classe `Cercle`. Ce contrôle est réalisé différemment suivant que le cercle est à créer ou à agrandir (*voir les méthodes* `créer()` *et* `agrandir()` *ci-dessus*).

- Soit le rayon n'est pas encore connu, et la vérification s'effectue dès la saisie de la valeur. C'est ce que réalise la méthode suivante :

```
   private int rayonVérifié() {
    int tmp;
    do  {
       System.out.print(" Rayon           :  ");
        tmp = Lire.i();
     } while ( tmp < 0 || tmp > TailleEcran) ;
   return tmp;
   }
```

- Soit le rayon est déjà connu, auquel cas la vérification est réalisée à partir de la valeur passée en paramètre de la méthode :

```
  private int rayonVérifié (int tmp) {
   if (tmp < 0) return 0;
     else if ( tmp > TailleEcran) return TailleEcran ;
     else return tmp;
  }
```

Les méthodes `rayonVérifié()` sont appelées **méthodes d'implémentation** car elles sont déclarées en mode privé. Leur existence n'est connue d'aucune autre classe. Seules les méthodes de la classe `Cercle` peuvent les exploiter, et elles ne sont pas directement exécutables par l'application. Elle sont cependant très utiles à l'intérieur de la classe où elles sont définies (*voir les sections « Les constructeurs » et « L'héritage »*).

Remarquons, en outre, que nous venons de définir deux méthodes portant le nom `rayonVérifié()`. Le langage Java n'interdit pas la définition de méthodes portant le même nom. Dans cette situation, on dit que ces méthodes sont **surchargées** (*voir la section « La surcharge de constructeurs »*).

Les constructeurs

Grâce aux différents niveaux de protection et aux méthodes contrôlant l'accès aux données, il devient possible de construire des outils appropriés aux objets manipulés.

Cependant, la protection des données d'une classe passe aussi par la notion de constructeurs d'objets. En effet, les constructeurs sont utilisés pour initialiser correctement les données d'un objet au moment de la création de l'objet en mémoire.

Le constructeur par défaut

Le langage Java définit, pour chaque classe construite par le programmeur, un constructeur par défaut. Celui-ci initialise, lors de la création d'un objet, toutes les données de cet objet à `0` pour les entiers, à `0.0` pour les réels, à `'\0'` pour les caractères et à `null` pour les `String` ou autres types structurés.

Le constructeur par défaut est appelé par l'opérateur `new` lors de la réservation de l'espace mémoire. Ainsi, lorsque nous écrivons :

```
Cercle C = new Cercle();
```

nous utilisons le terme `Cercle()`, qui représente en réalité le constructeur par défaut (il ne possède pas de paramètre) de la classe `Cercle`.

Un constructeur est une méthode, puisqu'il y a des parenthèses `()` derrière son nom d'appel, qui porte le nom de la classe associée au type de l'objet déclaré.

Définir le constructeur d'une classe

L'utilisation du constructeur par défaut permet d'initialiser systématiquement les données d'une classe. L'initialisation proposée peut parfois ne pas être conforme aux valeurs demandées par le type.

Dans ce cas, le langage Java offre la possibilité de définir un constructeur propre à la classe de l'objet utilisé. Cette définition est réalisée en écrivant une méthode portant le même nom que sa classe. Les instructions qui la composent permettent d'initialiser les données de la classe, conformément aux valeurs demandées par le type choisi.

Par exemple, le constructeur de la classe `Cercle` peut s'écrire de la façon suivante :

```
public Cercle()    {
   System.out.print(" Position en x :  ");
   x = Lire.i();
   System.out.print(" Position en y :  ");
   y = Lire.i();
   r = rayonVérifié();
}
```

En observant la structure du constructeur `Cercle()`, nous constatons qu'un constructeur n'est pas typé. Aucun type de retour n'est placé dans son en-tête. Mais attention ! le fait d'écrire l'en-tête `public void Cercle()` ou encore `public int Cercle()` a pour résultat de créer une simple méthode, qui a pour nom `Cercle()` et qui n'est pas celle appelée par l'opérateur `new`. Il ne s'agit donc pas d'un constructeur.

Une fois correctement défini, le constructeur est appelé par l'opérateur `new`, comme pour le constructeur par défaut. L'instruction :

```
Cercle A = new Cercle();
```

fait appel au constructeur défini ci-dessus. Le programme exécuté demande, dès la création de l'objet `A`, de saisir les données le concernant, avec une vérification concernant la valeur du rayon grâce à la méthode `rayonVérifié()`. De cette façon, l'application est sûre d'exploiter des objets dont la valeur est valide dès leur initialisation.

Remarquons que :

- Lorsqu'un constructeur est défini par le programmeur, le constructeur proposé par défaut par le langage Java n'existe plus.
- La méthode `créer()` et le constructeur ainsi définis ont un rôle identique. La méthode `créer()` devient par conséquent inutile.

La surcharge de constructeurs

Le langage Java permet la définition de plusieurs constructeurs, ou méthodes, à l'intérieur d'une même classe, du fait que la construction des objets peut se réaliser de différentes façons. Lorsqu'il existe plusieurs constructeurs, on dit que le constructeur est **surchargé**.

Dans la classe `Cercle`, il est possible de définir deux constructeurs supplémentaires :

```
public Cercle(int centrex, int centrey)        {
  x = centrex ;
  y = centrey;
}
public Cercle(int centrex, int centrey, int rayon)        {
  this( centrex, centrey) ;
  r = rayonVérifié(rayon);
}
```

Pour déterminer quel constructeur doit être utilisé, l'interpréteur Java regarde, lors de son appel, la liste des paramètres définis dans chaque constructeur. La construction des trois objets `A`, `B` et `C` suivants fait appel aux trois constructeurs définis précédemment :

```
Cercle A = new Cercle();
Cercle B = new Cercle(10, 10);
Cercle c = new Cercle(10, 10, 30);
```

Lors de la déclaration de l'objet `A`, le constructeur appelé est celui qui ne possède pas de paramètre (le constructeur par défaut, défini à la section « Définir le constructeur d'une classe »), et les valeurs du centre et du rayon du cercle `A` sont celles saisies au clavier par l'utilisateur.

La création de l'objet `B` fait appel au constructeur qui possède deux paramètres de type entier. Les valeurs du centre et du rayon du cercle `B` sont donc celles passées en paramètre du constructeur, soit (`10`, `10`) pour (`B.x`, `B.y`). Aucune valeur n'étant précisée pour le rayon, `B.r` est automatiquement initialisé à `0`.

Le mot-clé this

La création de l'objet C est réalisée par le constructeur qui possède trois paramètres entiers. Ces paramètres permettent l'initialisation de toutes les données définies dans la classe Cercle.

Remarquons que, grâce à l'instruction this(centrex, centrey) , le constructeur possédant deux paramètres est appelé à l'intérieur du constructeur possédant trois paramètres.

Le mot-clé this() représente l'appel au second constructeur de la même classe possédant deux paramètres entiers, puisque this() est appelé avec deux paramètres entiers. Il permet l'utilisation du constructeur précédent pour initialiser les coordonnées du centre avant d'initialiser correctement la valeur du rayon grâce à la méthode rayonVérifié(rayon), qui est elle-même surchargée. Comme pour les constructeurs, le compilateur choisit la méthode rayonVérifié(), définie avec un paramètre entier.

Pour finir, remarquons que le terme this() doit toujours être placé comme première instruction du constructeur qui l'utilise.

L'héritage

L'héritage est le dernier concept fondamental de la programmation objet étudiée dans ce chapitre. Ce concept permet la réutilisation des fonctionnalités d'une classe, tout en apportant certaines variations, spécifiques de l'objet héritant.

Avec l'héritage, les méthodes définies pour un ensemble de données sont réutilisables pour des variantes de cet ensemble. Par exemple, si nous supposons qu'une classe Forme définisse un ensemble de comportements propres à toute forme géométrique, alors :

- Ces comportements peuvent être réutilisés par la classe Cercle, qui est une forme géométrique particulière. Cette réutilisation est effectuée sans avoir à modifier les instructions de la classe Forme.
- Il est possible d'ajouter d'autres comportements spécifiques des objets Cercle. Ces nouveaux comportements sont valides uniquement pour la classe Cercle et non pour la classe Forme.

La relation « est un »

En pratique, pour déterminer si une classe B **hérite** d'une classe A, il suffit de savoir s'il existe une relation « **est un** » entre B et A. Si tel est le cas, la syntaxe de déclaration est la suivante :

```
class B extends A        {
// données et méthodes de la classe B
}
```

Dans ce cas, on dit que :

- B est une **sous-classe** de A ou encore une **classe dérivée** de A.
- A est une **super-classe** ou encore une **classe de base**.

Un cercle « est une » forme géométrique

En supposant que la classe `Forme` possède des caractéristiques communes à chaque type de forme géométrique (les coordonnées d'affichage à l'écran, la couleur, etc.), ainsi que des comportements communs (afficher, déplacer, etc.), la classe `Forme` s'écrit de la façon suivante :

```
public class Forme       {
 protected int x, y ;
 private couleur ;

 public  Forme() {            // Le constructeur de la classe Forme
   System.out.print(" Position en x :  ");
   x = Lire.i();
   System.out.print(" Position en y :  ");
   y = Lire.i();
   System.out.print(" Couleur de la forme :  ");
   couleur = Lire.i();
 }

 public void afficher()  { //Affichage des données de la classe
  System.out.println(" Position en " + x + ", " + y);
  System.out.println(" Couleur : " + couleur);
 }

 public void déplacer(int nx, int ny) {    // Déplace les coordonnées de la
  x = nx;                                  // forme en (nx, ny) passées en
  y = ny;                                  // paramètre de la fonction
 }
} // Fin de la classe Forme
```

Sachant qu'un objet `Cercle` « est une » forme géométrique particulière, la classe `Cercle` hérite de la classe `Forme` en écrivant :

```
public class Cercle extends Forme     {
 private int r ;          //  rayon

  public Cercle()    {    // Le constructeur de la classe Cercle
   System.out.print(" Rayon            :  ");
   r = rayonVérifié();
}
private int rayonVérifié() {
// Voir la section Des méthodes invisibles
}
private int rayonVérifié (int tmp) {
// Voir la section Des méthodes invisibles
}

 public void afficher() { //Affichage des données de la classe
  super.afficher();
  System.out.println(" Rayon : " + r);
```

```
   }

  public double périmètre() {
   // voir la section "La classe descriptive du type Cercle" du chapitre
   //"Les classes et les objets"
 }

  public void agrandir(int nr) {               // Augmente la valeur courante du
   r = rayonVérifié(r + nr);                   // rayon avec la valeur passée en
  }                                            // paramètre
} // Fin de la classe Cercle
```

Un cercle est une forme géométrique (`Cercle extends Forme`) qui possède un rayon (`private int r`) et des comportements propres aux cercles, soit, par exemple, le calcul du périmètre (`périmètre()`) ou encore la modification de sa taille (`agrandir()`). Un cercle peut être déplacé, comme toute forme géométrique. Les méthodes de la classe `Forme` restent donc opérationnelles pour les objets `Cercle`.

En examinant de plus près les classes `Cercle` et `Forme`, nous remarquons que :

- La notion de constructeur existe aussi pour les classes dérivées (*voir la section « Le constructeur d'une classe héritée »*).
- Les données `x`, `y` sont déclarées `protected` (*voir la section « La protection des données héritées »*).
- La fonction `afficher()` existe sous deux formes différentes dans la classe `Forme` et la classe `Cercle`. Il s'agit là du concept de polymorphisme (*voir la section « Le polymorphisme »*).

Le constructeur d'une classe héritée

Les classes dérivées possèdent leurs propres constructeurs, qui sont appelés par l'opérateur `new`, comme dans :

```
Cercle A = new Cercle( );
```

Pour construire un objet dérivé, il est indispensable de construire d'abord l'objet associé à la classe mère. Pour construire un objet `Cercle`, nous devons définir ses coordonnées et sa couleur. Le constructeur de la classe `Cercle` doit appeler le constructeur de la classe `Forme`.

Par défaut, s'il n'y a pas d'appel explicite au constructeur de la classe supérieure, comme c'est le cas pour notre exemple, le compilateur recherche de lui-même le constructeur par défaut (sans paramètre) de la classe supérieure. En construisant l'objet `A`, l'interpréteur exécute aussi le constructeur par défaut de la classe `Forme`. L'ensemble des données du cercle (`x`, `y`, `couleur` et `r`) est alors correctement initialisé par saisie des valeurs au clavier.

Ce fonctionnement pose problème lorsqu'il n'existe pas de constructeur par défaut. Supposons que nous remplacions le constructeur de la classe `Forme` par :

```
public Forme(int nx, int ny) {         // Le nouveau constructeur de la
  x = nx ;                             // classe Forme
```

```
    y = ny ;
    couleur = 0;
  }
```

Dans cette situation, lors de la construction de l'objet `A`, le compilateur recherche le constructeur par défaut de la classe supérieure, soit `Forme()` sans paramètre. Ne le trouvant pas, il annonce une erreur du type `no constructor matching Forme() found in class Forme`.

Le mot-clé super

Pour éviter ce type d'erreur, la solution consiste à appeler directement le constructeur de la classe mère depuis le constructeur de la classe :

```
public Cercle(int xx, int yy)       {    // Le constructeur de la classe Cercle
   super(xx, yy);
   System.out.print(" Rayon            :  ");
   r = rayonVérifié();
}
```

De cette façon, comme le terme `super()`, qui représente le constructeur de la classe supérieure possèdant deux entiers en paramètres, l'interpréteur peut finalement construire l'objet `A` (`Cercle A = new Cercle(5, 5)`), par appel du constructeur de la classe `Forme` à l'intérieur du constructeur de la classe `Cercle`.

Remarquons que le terme `super` est obligatoirement la première instruction du constructeur de la classe dérivée. La liste des paramètres (deux `int`) permet de préciser au compilateur quel est le constructeur utilisé en cas de surcharge de constructeurs.

La protection des données héritées

En héritant de la classe `A`, la classe `B` hérite des données et méthodes de la classe `A`. Cela ne veut pas forcément dire que la classe `B` ait accès à toutes les données et méthodes de la classe `A`. En effet, héritage n'est pas synonyme d'accessibilité.

Lorsqu'une donnée de la classe supérieure est déclarée en mode `private`, la classe dérivée ne peut ni consulter ni modifier directement cette donnée héritée. L'accès ne peut se réaliser qu'au travers des méthodes de la classe supérieure.

Pour notre exemple, la donnée `couleur` étant déclarée `private` dans la classe `Forme`, le constructeur suivant génère l'erreur `variable couleur in class Forme not accessible from class Cercle`.

```
public Cercle(int xx, int yy)   {   // Le constructeur de la classe Cercle
   super(xx, yy);
   couleur = 20 ;
   r = 10;
}
```

Il est possible, grâce à la protection `protected`, d'autoriser l'accès en consultation et modification des données de la classe supérieure. Toutes les données de la classe `A` sont alors accessibles depuis la classe `B`, mais pas depuis une autre classe.

Dans notre exemple, si la donnée `couleur` est déclarée `protected` dans la classe `Forme`, alors le constructeur de la classe `Cercle` peut modifier sa valeur.

Le polymorphisme

La notion de polymorphisme découle directement de l'héritage. Par polymorphisme, il faut comprendre qu'une méthode peut se comporter différemment suivant l'objet sur lequel elle est appliquée.

Lorsqu'une même méthode est définie à la fois dans la classe mère et dans la classe fille, l'exécution de la forme (méthode) choisie est réalisée en fonction de l'objet associé à l'appel et non plus suivant le nombre et le type des paramètres, comme c'est le cas lors de la surcharge de méthodes à l'intérieur d'une même classe.

Pour notre exemple, la méthode `afficher()` est décrite dans la classe `Forme` et dans la classe `Cercle`. Cette double définition ne correspond pas à une véritable surcharge de fonctions. Ici, les deux méthodes `afficher()` sont définies sans aucun paramètre. Le choix de la méthode ne peut donc s'effectuer sur la différence des paramètres. Il est effectué par rapport à l'objet sur lequel la méthode est appliquée. Observons l'exécution du programme suivant :

```
public class FormerDesCercles           {
 public static void main(String [] arg)    {
  Cercle A  = new Cercle(5, 5);
  A.afficher();
  Forme F = new Forme (10, 10, 3);
  F.afficher();
 }
}
```

L'appel du constructeur de l'objet `A` nous demande de saisir la valeur du rayon :

```
Rayon             :  7
```

La méthode `afficher()` est appliquée à `A`. Puisque `A` est de type `Cercle`, l'affichage correspond à celui réalisé par la méthode définie dans la classe `Cercle`, soit :

```
Position en 5, 5
Couleur : 20
Rayon : 7
```

La forme `F` est ensuite créée puis affichée à l'aide la méthode `afficher()` de la classe `Forme`, `F` étant de type `Forme` :

```
Position en 10, 1
Couleur : 3
```

Remarquons que, lorsqu'une méthode héritée est définie une deuxième fois dans la classe dérivée, l'héritage est supprimé. Le fait d'écrire `A.afficher()` ne permet plus d'appeler directement la méthode `afficher()` de la classe `Forme`.

Pour appeler la méthode définie dans la classe supérieure, la solution consiste à utiliser le terme `super`, qui recherche la méthode à exécuter en remontant dans la hiérarchie.

Dans notre exemple, `super.afficher()` permet d'appeler la méthode `afficher()` de la classe `Forme`.

Grâce à cette technique, si la méthode d'affichage pour une `Forme` est transformée, cette transformation est automatiquement répercutée pour un `Cercle`.

Résumé

Lorsque l'interpréteur Java rencontre le mot-clé **`static`** devant une variable (**variable de classe**), il réserve un seul et unique emplacement mémoire pour cette variable. Si ce mot-clé est absent, l'interpréteur peut construire en mémoire la variable déclarée non `static` (**variable d'instance**) en plusieurs exemplaires. Cette présence ou cette absence du mot-clé `static` permet de différencier les variables des objets.

Les objets sont définis en mémoire par l'intermédiaire d'une **adresse** (**référence**). Lorsqu'un objet est passé en paramètre d'une fonction, la valeur passée au paramètre formel est l'adresse de l'objet. De cette façon, si la méthode transforme les données du paramètre formel, elle modifie aussi les données de l'objet effectivement passé en paramètre. Ainsi, tout objet passé en paramètre d'une méthode voit, en sortie de la méthode, ses données transformées par la méthode. Ce mode de transmission des données est appelé **passage** de paramètres **par référence**.

L'objectif principal de la programmation objet est d'écrire des programmes qui contrôlent par eux-mêmes le bien-fondé des opérations qui leur sont appliquées. Ce contrôle est réalisé grâce au principe d'**encapsulation** des données. Par ce terme, il faut comprendre que les données d'un objet sont protégées, de la même façon qu'un médicament est protégé par la fine capsule qui l'entoure. L'encapsulation passe par le **contrôle des données** et des comportements de l'objet à travers les niveaux de **protection**, l'**accès** contrôlé aux données et la notion de **constructeur** de classe.

Le langage Java propose trois niveaux de protection, `public`, `private` et `protected`. Lorsqu'une donnée est totalement protégée (`private`), elle ne peut être modifiée que par les méthodes de la classe où la donnée est définie.

On distingue les méthodes qui consultent la valeur d'une donnée sans pouvoir la modifier (**accesseur en consultation**) et celles qui modifient après contrôle et validation la valeur de la donnée (**accesseur en modification**).

Les constructeurs sont des méthodes particulières, déclarées uniquement `public`, qui portent le même nom que la classe où ils sont définis. Ils permettent le contrôle et la validation des données dès leur initialisation.

Par défaut, si aucun constructeur n'est défini dans une classe, le langage Java propose un constructeur par défaut, qui initialise toutes les données de la classe à `0` ou à `null`, si les données sont des objets. Si un constructeur est défini, le constructeur par défaut n'existe plus.

L'**héritage** permet la réutilisation des objets et de leur comportement, tout en apportant de légères variations. Il se traduit par le principe suivant : on dit qu'une classe `B` hérite d'une classe `A` (`B` étant une sous-classe de `A`) lorsqu'il est possible de mettre la relation « est un » entre `B` et `A`.

De cette façon, toutes les méthodes, ainsi que les données déclarées `public` ou `protected`, de la classe `A` sont applicables à la classe `B`. La syntaxe de déclaration d'une sous-classe est la suivante :

```
class B extends A        {
// données et méthodes de la classe B
}
```

Le projet « Gestion d'un compte bancaire »

Encapsuler les données d'un compte bancaire

La protection `privée` et l'accès aux données

a. Déclarez toutes les variables d'instance des types `Compte` et `LigneComptable` en mode `private`. Que se passe-t-il lors de la phase de compilation de l'application `Projet` ?

Pour remédier à cette situation, la solution est de construire des méthodes d'accès aux données de la classe `Compte` et `LigneComptable`. Ces méthodes ont pour objectif de fournir au programme appelant la valeur de la donnée recherchée. Par exemple, la fonction `quelTypeDeCompte()` suivante fournit en retour le type du compte recherché :

```
public String quelTypeDeCompte()     {
    return typeCpte;
}
```

b. Écrivez, suivant le même modèle, toutes les méthodes d'accès aux données `val_courante`, `taux`, `numéroCpte`, etc.

c. Modifiez l'application `Projet` et la classe `Compte` de façon à pouvoir accéder aux données `numéroCpte` de la classe `Compte` et aux valeurs de la classe `LigneComptable`.

Le contrôle des données

L'encapsulation des données permet le contrôle de la validité des données saisies pour un objet. Un compte bancaire ne peut être que de trois types : `Epargne`, `Courant` ou `Joint`. Il est donc nécessaire, au moment de la saisie du `type` du compte, de contrôler l'exactitude du type entré. La méthode `contrôleType()` suivante réalise ce contrôle :

```
private String contrôleType()    {
 char tmpc;
 String tmpS = "Courant";
 do {
  System.out.print("Type du compte [Types possibles : C(ourant), ");
  System.out.print("J(oint), E(pargne)] : ");
  tmpc = Lire.c();
 } while ( tmpc != 'C' && tmpc != 'J' && tmpc != 'E');
  switch (tmpc) {
   case 'C' : tmpS = "Courant";
              break;
```

```
    case 'J' : tmpS = "Joint";
               break;
    case 'E' : tmpS = "Epargne";
               break;
  }
  return tmpS;
}
```

À la sortie de la fonction, nous sommes certains que le type retourné correspond aux types autorisés par le cahier des charges.

a. Dans la classe `Compte`, sachant que la valeur initiale ne peut être négative à la création d'un compte, écrivez la méthode `contrôleValinit()`.

b. Dans la classe `LigneComptable`, écrivez les méthodes `contrôleMotif()` et `contrôleMode()`, qui vérifient respectivement le motif (`Salaire`, `Loyer`, `Alimentation`, `Divers`) et le mode (`CB`, `Virement`, `Chèque`) de paiement pour une ligne comptable.

✔ Pour contrôler la validité de la date, voir la section « *Le projet…* » du chapitre 10, « *Collectionner un nombre indéterminé d'objets* ».

c. Modifiez les méthodes `créerCpte()` et `créerLigneComptable()` de façon que les données des classes `Compte()` et `LigneComptable()` soient valides.

Les constructeurs de classe

Les constructeurs `Compte()` et `LigneComptable()` s'inspirent pour une grande part des méthodes `créerCpte()` et `créerLigneComptable()`.

a. Remplacez directement `créerCpte()` par `Compte()`. Que se passe-t-il lors de l'exécution du programme ?

b. Déplacez l'appel au constructeur dans l'option `1`, de façon à construire l'objet au moment de sa création. Que se passe-t-il en phase de compilation ? Pourquoi ?

c. Utilisez la notion de surcharge de constructeur pour construire un objet `C` de deux façons :

- Les valeurs initiales du compte sont passées en paramètre.
- Les valeurs initiales sont saisies au clavier, comme le fait la méthode `créerCpte()`.

d. À l'aide de ces deux constructeurs, modifiez l'application `Projet` de façon à pouvoir l'exécuter correctement.

Comprendre l'héritage

Protection des données héritées

Sachant qu'un compte d'épargne est un compte bancaire ayant un taux de rémunération,

a. Écrivez la classe `CpteEpargne` en prenant soin de déclarer la nouvelle donnée en mode `private`.

b. Modifiez le type `Compte` de façon à supprimer tout ce qui fait appel au compte d'épargne (donnée et méthodes).

Un compte d'épargne modifie la valeur courante par le calcul des intérêts, en fonction du taux d'épargne. Il ne peut ni modifier son numéro, ni son type.

c. Quels modes de protection doit-on appliquer aux différentes données héritées de la classe `Compte` ?

Le contrôle des données d'un compte d'épargne

Sachant que le taux d'un compte d'épargne ne peut être négatif, écrivez la méthode `contrôleTaux()`.

Le constructeur d'une classe dérivée

En supposant que le constructeur de la classe `CpteEpargne` s'écrive de la façon suivante :

```
public CpteEpargne() {
  super("Epargne");
  taux = contrôleTaux();
}
```

a. Recherchez à quel constructeur de la classe `Compte` fait appel `CpteEpargne()`. Pourquoi ?

b. Modifiez ce constructeur de façon que la donnée `typeCpte` prenne la valeur `Epargne`.

Le polymorphisme

De la méthode `afficherCpte()` :

a. Dans la classe `CpteEpargne`, écrivez la méthode `afficherCpte()`, sachant qu'afficher les données d'un compte d'épargne revient à afficher les données d'un compte, suivi du taux d'épargne.

De l'objet `C`, déclaré de type `Compte` :

b. Dans l'application `Projet`, modifiez l'option `1`, de façon à demander à l'utilisateur s'il souhaite créer un compte simple ou un compte d'épargne. Selon la réponse, construisez l'objet `C` en appelant le constructeur approprié.

Partie 3

Les outils et techniques orientés objet

9

Collectionner un nombre fixe d’objets

Comme nous l’avons observé tout au long de cet ouvrage, l’atout principal de l’ordinateur est sa capacité à manipuler un grand nombre de données pour en extraire de nouvelles informations. Or, les structures de stockage étudiées jusqu’ici, telles que variables ou objets, ne permettent pas d’appliquer de traitements systématiques sur des ensembles de valeurs.

C’est pourquoi nous étudions dans ce chapitre une nouvelle structure de données, les tableaux, qui permettent le stockage d’un nombre fini de valeurs.

Dans un premier temps, nous étudions « Les tableaux à une dimension » et observons comment les déclarer et les manipuler. Pour mieux comprendre la manipulation de ces structures, nous analysons ensuite, à la section « Quelques techniques utiles », différentes techniques de programmation appliquées aux tableaux à une dimension, telles que la recherche d’une valeur dans un tableau ou le tri d’un tableau.

Pour finir, nous examinons, à la section « Les tableaux à deux dimensions », comment construire et manipuler des tableaux bidimensionnels à travers un exemple d’affichage de formes géométriques.

Les tableaux à une dimension

L’étude du chapitre 1, « Stocker une information », montre que, pour manipuler plusieurs valeurs à l’intérieur d’un programme, vous devez déclarer autant de variables que de valeurs à traiter. Ainsi, pour stoker les huit notes d’un élève donné, la technique consiste à déclarer huit variables, comme suit :

```
double note1, note2, note3, note4, note5, note6, note7, note8;
```

Le fait de déclarer autant de variables qu'il y a de valeurs présente les inconvénients suivants :

- Si le nombre de notes est modifié, il est nécessaire de :
  - Déclarer de nouvelles variables.
  - Placer ces variables dans le programme, afin de les traiter en plus des autres notes.
  - Compiler à nouveau le programme pour que l'interpréteur puisse prendre en compte ces modifications.
- Il faut trouver un nom de variable pour chaque valeur traitée. Imaginez déclarer 1 000 variables portant un nom différent !

Ces inconvénients majeurs sont résolus grâce aux tableaux. En effet, les tableaux sont des structures de données qui regroupent sous un même nom de variable un nombre donné de valeurs de même type. Les tableaux sont proposés par tous les langages de programmation. Ils sont construits par assemblage d'une suite finie de cases mémoire, comme illustré à la Figure 9-1.

Chaque case représente l'espace mémoire nécessaire au stockage d'une et une seule valeur. Remarquez que les cases sont réservées en mémoire de façon contiguë.

Déclarer un tableau

Comme toute variable utilisée dans un programme, un tableau doit être déclaré afin de

- donner un nom à l'ensemble des valeurs à regrouper ;
- définir la taille du tableau de façon à préciser le nombre de valeurs à regrouper ;
- déterminer le type de valeur à mémoriser.

La syntaxe de déclaration d'un tableau est la suivante :

```
TypeDuTableau [] nomDuTableau ;
nomDuTableau = new TypeDuTableau [tailleDuTableau] ;
```

En plaçant dans la première instruction les crochets `[]` entre le type et le nom de la variable, vous indiquez au compilateur que la variable `nomDuTableau` représente un tableau. À cette étape, le compilateur réserve un espace mémoire portant le nom du tableau. Cet espace mémoire est susceptible de contenir l'adresse de la première case du tableau.

Ensuite, dans la seconde instruction, l'opérateur `new` réserve autant de cases mémoire consécutives qu'il est indiqué entre les `[]` situés en fin d'instruction, soit `tailleDuTableau`.

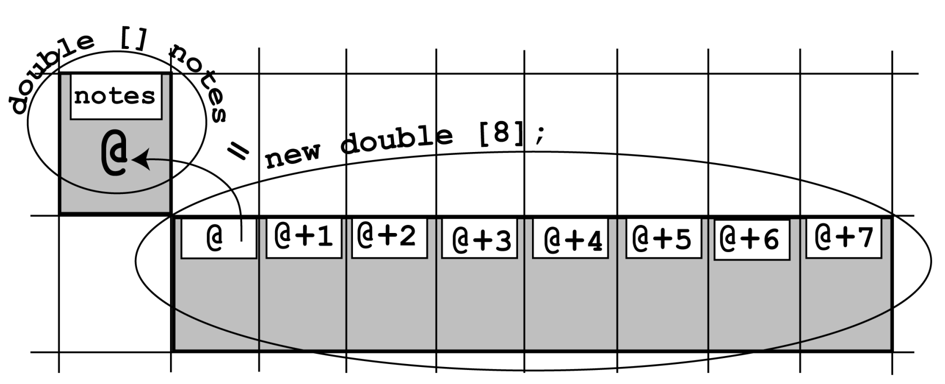


Figure 9-1.

L'opérateur `new` *réserve le nombre de cases mémoire demandé (*`[8]`*) et mémorise l'adresse de la première case mémoire dans la variable* `notes` *grâce au signe d'affectation.*

L'opérateur `new` détermine enfin l'adresse de la première case du tableau et la stocke, grâce au signe d'affectation, dans la case `nomDuTableau` créée à l'étape précédente.

Exemple : Déclarer un tableau de huit notes

```
double [] notes ;
notes = new double[8] ;
```

Ces deux instructions réalisent la déclaration d'un tableau ayant pour nom `note`. Il est composé de `8` cases mémoire pouvant stocker des valeurs de type `double` (*voir Figure 9-1*).

Autres exemples de déclaration

```
// déclarer un tableau de 5 entiers
int [] valeur ;
valeur = new int[5];
// déclarer un tableau de 30 réels de simple précision
float [] reel ;
reel = new float[30];
// déclarer un tableau de 80 caractères
char [] mot ;
mot = new char[80];
```

Remarques

- Le nombre de cases réservées correspond au nombre maximal de valeurs à traiter. Lorsque la taille du tableau est fixée après exécution de l'opérateur `new`, il n'est plus possible de la modifier en cours d'exécution du programme.

  Cependant, il est possible de ne pas fixer définitivement la taille du tableau avant compilation en plaçant une variable entre les `[]` au lieu d'une valeur numérique. En effet, il suffit d'écrire :

```
double []    notes;
int     nbNotes ;
System.out.println("Combien voulez-vous saisir de notes : ");
nbNotes = Lire.i() ;
notes = new double[nbNotes];
```

  De cette façon, l'utilisateur saisit le nombre de valeurs qu'il souhaite traiter avant la réservation effective des espaces mémoire par l'opérateur `new`. Le programme peut donc voir la taille du tableau varier d'une exécution à l'autre.

- Les tableaux sont des objets. En effet, les tableaux sont définis à l'aide d'une adresse déterminée par l'opérateur `new`. Les tableaux sont donc des objets, au même titre que les `String` et autres objets définis aux chapitres précédents.

  Les objets sont caractérisés par leurs données et les méthodes qui leur sont applicables. Une donnée caractéristique des tableaux est leur taille, c'est-à-dire le nombre de cases. Ainsi, pour connaître la taille d'un tableau, il suffit de placer le terme `.length` derrière le nom du tableau.

Par exemple, l'instruction suivante :

```
System.out.print("Nombre de notes = "+ notes.length) ;
```

affiche à l'écran `Nombre de notes = 8`.

- L'instruction de déclaration :

```
double [] notes = new double[8];
```

est équivalente à la suite d'instructions :

```
double [] notes ;
notes = new double[8] ;
```

Manipuler un tableau

Un tableau est un ensemble de cases mémoire. Chaque case constituant un élément du tableau est identique à une variable. Il est possible de manipuler chaque case du tableau de façon à

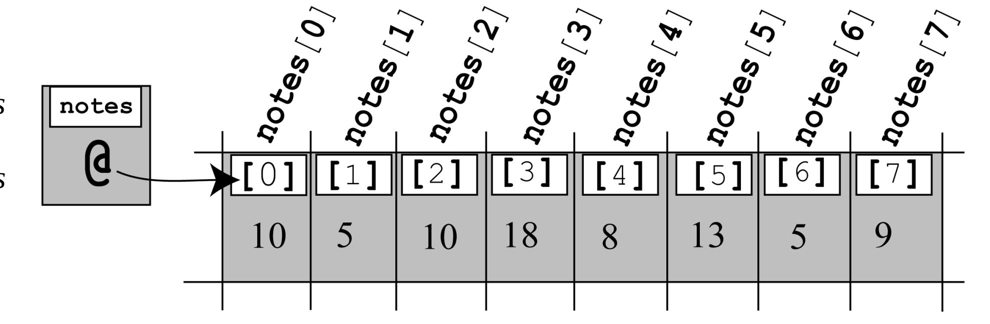
- placer une valeur dans une case du tableau à l'aide de l'affectation ;
- utiliser un élément du tableau dans le calcul d'une expression mathématique ;
- afficher un élément du tableau.

Accéder aux éléments d'un tableau

Sachant que `nomDuTableau[0]` représente la première case du tableau, l'accès à la `n`*ième* case s'écrit `nomDuTableau[n]`.

Figure 9-2.

`Note` *est le nom du tableau, et les notes* `10, 5, …, 9` *sont des valeurs placées à l'aide du signe d'affectation dans les cases numérotées respectivement* `0, 1, …, 7` *(indices).*



Par exemple, l'instruction

```
note[0] = 10 ;
```

mémorise la première note d'un étudiant dans la première case du tableau (`notes[0]`). De la même façon, la deuxième note est stockée grâce à l'affectation

```
note[1] = 5 ;
```

Et ainsi de suite, jusqu'à stoker la huitième et dernière note à l'aide de l'instruction

```
note[7] = 9 ;
```

Les valeurs placées entre les crochets `[]` sont appelées les **indices** du tableau. Remarquez que la première case du tableau est numérotée à partir de `0` et non de `1` (*voir Figure 9-2*). L'indice du tableau varie donc entre `0` et `length-1`. Les éléments d'un tableau étant ordonnés grâce aux indices, il est possible d'y accéder à l'aide de constructions itératives (boucle `for`) , comme le montre l'exemple suivant.

Exemple : Extrait d'un programme

```
System.out.print("Combien de notes voulez-vous saisir ; ");
int nombre = Lire.i();
notes = new double [nombre];
for (int i = 0; i < notes.length; i++) {
  System.out.print("Entrer la note  n° "+ (i + 1)+ " :  ");
  notes[i] = Lire.d();
}
```

Exemple : Résultat de l'exécution

Les caractères grisés sont des valeurs choisies par l'utilisateur.

```
Combien de notes voulez-vous saisir : 4
Entrer la note  n° 1 : 14
Entrer la note  n° 2 : 10
Entrer la note  n° 3 : 12
Entrer la note  n° 4 : 8
```

Une fois le nombre de notes déterminé grâce aux deux premières instructions, le programme entre dans une boucle `for`. La variable `i` correspond au compteur de boucle. Elle varie entre `0` et `notes.length-1` (soit `3`), puisque la condition de continuation précise que `i` doit être strictement inférieure à `notes.length` (soit `4`).

À chaque tour de boucle, la variable `i` prend la valeur de l'indice du tableau (`notes[i]`). Les valeurs saisies au clavier sont alors placées une à une dans chaque case du tableau.

Parce qu'il n'est pas courant de compter des valeurs à partir de `0`, l'affichage demandant d'entrer une note débute à `1`, et non à `0`, grâce à l'expression `(i+1)` placée dans la méthode `System.out.print()`. Il ne s'agit là que d'un artifice de présentation, la première note étant stockée en réalité en `note[0]`.

Remarquez que l'utilisation de la donnée `length` permet d'éviter tout problème de dépassement de taille. En effet, si un tableau est composé de quatre cases, il n'est pas possible de placer une valeur à l'indice 4 ou 5. Le fait d'écrire `notes[4]` génère une erreur d'exécution du type : `java.lang.ArrayIndexOutOfBoundsException`, qui montre que l'interpréteur Java a détecté que l'indice du tableau était en dehors des limites définies au moment de sa création.

Initialiser un tableau

Lors de la déclaration d'un tableau, il est possible de l'initialiser directement de la façon suivante :

```
double [] notes = {10, 12.5, 5, 8.5, 16, 0, 13, 7} ;
```

Les cases mémoire sont réservées et initialisées, dans l'ordre, à l'aide des valeurs placées entre les `{}` et séparées par des virgules. De cette façon, le tableau `notes` contient les valeurs suivantes :

notes[0]	vaut	10	notes[4]	vaut	16
notes[1]	vaut	12.5	notes[5]	vaut	0
notes[2]	vaut	5	notes[6]	vaut	13
notes[3]	vaut	8.5	notes[7]	vaut	7

Remarquez que la donnée `notes.length` prend automatiquement la valeur 8.

Les tableaux et les opérations arithmétiques

La somme, la soustraction, la division ou la multiplication directes de deux tableaux sont des opérations impossibles. En effet, chaque opération doit être réalisée élément par élément, comme le montre le tableau suivant :

Correcte	Impossible

`int [] tab1 = new int[10] ;`
`int [] tab2 = new int[10];`
`int [] somme = new int[10];`
`for (i = 0 ; i < 10 ; i++)`
`somme[i] = tab1[i] + tab2[i];`
	`int [] tab1 = new int[10] ;`
	`int [] tab2 = new int[10];`
	`int [] somme = new int[10];`
	
	~~`somme = tab1 + tab2;`~~

Quelques techniques utiles

Le stockage et l'utilisation des données à travers la structure des tableaux offrent de nombreux avantages. Elles requièrent aussi certaines techniques de manipulation, qui sont développées ci-après.

La ligne de commande

Au cours du chapitre « Naissance d'un programme », vous avez dû admettre un certain nombre de termes du langage Java et, en particulier, la syntaxe de l'instruction suivante :

```
public static void main(String [] argument)
```

Cette instruction correspond à la définition de l'en-tête de la fonction `main()`. Vous êtes en mesure maintenant de déchiffrer chacun de ses termes pour en comprendre l'utilité :

- Le mot clé `public` précise au compilateur que la fonction `main()` est accessible depuis l'extérieur de la classe où elle est définie. En particulier, l'interpréteur Java peut y accéder pour l'exécuter.
- Le terme `static` explique que la fonction `main()` ne peut pas être copiée plusieurs fois en mémoire. Elle ne peut pas être associée à un objet ni être instanciée, c'est-à-dire qu'il n'est pas possible d'écrire `unObjet.main()`.
- La fonction `main()` ne fournit pas de résultat, et c'est pourquoi elle est définie comme `void`.

- Pour finir, elle possède, entre `()`, un paramètre défini comme tableau de type `String`. Ce paramètre est utilisé pour passer des données en **ligne de commande** lors du lancement de la commande d'exécution du programme.

Qu'est-ce qu'une ligne de commande ?

Une ligne de commande est écrite au clavier sous la forme d'une instruction précise. C'est un ordre transmis à l'ordinateur par l'utilisateur. Sous Unix, les commandes sont très utilisées. Elles le sont beaucoup moins sous Windows et sont inexistantes sous Mac OS. C'est pourquoi les utilisateurs de stations de travail Unix n'ont aucune difficulté à comprendre ce qu'est une commande, ce qui n'est pas le cas des utilisateurs de PC (sous Windows) ou de Macintosh.

Aujourd'hui, grâce aux écrans graphiques, l'utilisateur communique facilement avec l'ordinateur. Pour savoir ce que contient un dossier, il lui suffit d'ouvrir la fenêtre associée à ce dossier. Les ordres passés à l'ordinateur sont essentiellement des ordres générés par la souris au travers de fenêtres graphiques.

Les lignes de commande sont équivalentes, bien que moins conviviales, à cette communication graphique. Elles permettent surtout d'obtenir des résultats plus précis. Ainsi, les commandes :

- `ls *.java`, dans une fenêtre de commandes Unix,
- `dir *.java`, dans une fenêtre « commandes MSDOS »,

ont pour résultat d'afficher tous les noms de fichiers finissant par `.java` contenus dans le répertoire courant.

Plus précisément, remarquez qu'une commande s'écrit toujours de la façon suivante :

```
nomDeLaCommande paramètresEventuels
```

Le nom d'une commande correspond au nom du programme qui réalise l'action souhaitée. Les paramètres sont utilisés pour affiner son résultat. Dans notre exemple, `*.java` est un paramètre des commandes `ls` ou `dir`, qui permet d'expliquer à l'ordinateur que vous souhaitez voir s'afficher uniquement les noms de fichiers finissant par `.java`.

Passer des paramètres à un programme Java

De la même façon, comme expliqué à la section « Exécuter un programme » du chapitre introductif « Naissance d'un programme », l'exécution d'un programme Java en dehors d'un environnement de travail passe aussi par une commande dont la syntaxe est :

```
java nomdel'application
```

L'interpréteur Java autorise aussi la commande :

```
java nomdel'application p0 p1 p2… pN
```

Dans ce cas, les valeurs `p0`, `p1`, `p2`, … `pN`, toutes séparées par des espaces, sont considérées comme paramètres de la commande `java nomdel'application`. Ces derniers sont transmis à la fonction `main()` par l'intermédiaire du tableau de `String` défini en paramètre de la fonction.

Si l'en-tête est de la forme :

```
public static void main(String [] argument)
```

le paramètre `p0` est stocké en `argument[0]`, `p1` en `argument[1]`, … et `pN` en `argument[N]`. Les valeurs ainsi passées sont mémorisées sous forme de chaînes de caractères.

Exemple : Une commande qui calcule

Pour mieux comprendre cette transmission de valeurs, reprenons le corrigé de l'exercice 3 du Chapitre 6, « Fonctions, notions avancées », qui simule une calculette. Transformons ce programme de sorte qu'il puisse effectuer l'opération à partir de valeurs passées en paramètres lors de la commande d'exécution du programme. Supposons que cette commande s'écrive :

```
java Calculette 1 + 2
```

L'ordre des paramètres ainsi passés est important. En effet, nous devons traiter les paramètres de la fonction `main()` de la façon suivante :

- Les premier et troisième paramètres doivent être interprétés comme étant les valeurs numériques de l'opération à calculer.
- Le deuxième paramètre doit correspondre à l'opérateur.

Sachant cela, le programme s'écrit de la façon suivante :

Exemple : Le code source

```
public class Calculette {
 public static void main(String [] argument) {
  int a, b;
  char opérateur;
  double calcul;
     if (argument.length > 0) {
      a = Integer.parseInt(argument[0]);
       opérateur = argument[1].charAt(0);
       b = Integer.parseInt(argument[2]);
     }
  else {
    opérateur = menu();
    System.out.println("Entrer la premiere valeur ");
    a = Lire.i();
    System.out.println("Entrer la seconde valeur ");
    b = Lire.i();
  }
   calcul = calculer(a, b, opérateur );
   afficher(a, b, opérateur, calcul);
 }

 public static double calculer (int x, int y, char o) {
 // voir corrigé de l'exercice 3 du chapitre Fonctions, notions avancées
 }

 public static void afficher(int x, int y, char o, double r) {
```

```
// voir corrigé de l'exercice 3 du chapitre Fonctions, notions avancées
 }
 public static char menu() {
 // voir corrigé de l'exercice 3 du chapitre Fonctions, notions avancées
 }
}
```

Pour traiter les paramètres passés en ligne de commande, il est nécessaire de détecter si des paramètres ont été effectivement passés. Pour ce faire, l'idée est de regarder la taille du tableau `argument`, de la façon suivante :

- Si celle-ci n'est pas nulle (supérieure strictement à 0), cela signifie que le tableau contient des paramètres passés en ligne de commande. Dans ce cas, nous traitons chacun des arguments de sorte que le calcul puisse être effectué.

  Les éléments `argument[0]` et `argument[2]` contiennent par hypothèse les deux valeurs numériques. Or, celles-ci sont stockées sous forme de suites de caractères, le tableau `argument` étant de type `String`. Les valeurs doivent donc être « traduites » en format numérique. Comme nous souhaitons obtenir des valeurs entières, la méthode proposée par le langage Java a pour nom `Integer.parseInt()`. Ainsi, les instructions :

```
a = Integer.parseInt(argument[0]);
b = Integer.parseInt(argument[2]);
```

  permettent la traduction de la suite de caractères contenue dans `argument[0]` et `argument[2]` en valeurs numériques et de placer ces valeurs dans les variables `a` et `b` déclarées de type `int`. Compte tenu des paramètres passés en ligne de commande, les variables `a` et `b` ont donc pour valeurs respectives `1` et `2`.

  Le caractère correspondant à l'opérateur est stocké dans `argument[1]`. Nous devons le transformer en `char` puisqu'un opérateur est formé d'un seul caractère. Cette transformation est réalisée par une méthode de la classe `String`, appelée `charAt()`, qui retourne le caractère placé à la position spécifiée en paramètre. Ainsi, l'instruction :

```
opérateur = argument[1].charAt(0);
```

  place dans la variable `opérateur` le premier caractère du mot stocké dans `argument[1]`, soit, pour notre exemple, le caractère +.

- Si la taille du tableau `argument` est nulle, cela signifie qu'aucun paramètre n'a été transmis. Le bloc `else` est exécuté, et les valeurs sont saisies au clavier, comme cela était le cas en fin de correction de l'exercice 3 du Chapitre 6.

Pour finir, lorsque les valeurs choisies sont placées dans les variables `a`, `b` et `opérateur`, à l'aide des paramètres ou du clavier, le calcul de l'opération est réalisé, et le résultat est affiché. Dans l'exemple, vous obtenez :

```
java Calculette 1 + 2
1 + 2 = 3
```

Précisons en outre que cette commande doit être obligatoirement lancée dans le répertoire où se trouve le fichier `Calculette.class`.

Trier un ensemble de données

L'atout principal de l'ordinateur est sa faculté à traiter un très grand nombre de données en des temps très rapides. Ces traitements sont, par exemple, la recherche d'éléments dans un ensemble en suivant des contraintes choisies par l'utilisateur ou encore le tri d'éléments en fonction d'un critère déterminé.

Pour comprendre le fonctionnement interne de ces traitements, nous étudions ici l'algorithme du « tri par extraction simple », qui utilise les techniques de recherche d'un élément dans un ensemble de données, d'échange de valeurs et de tri.

Cahier des charges

L'objectif du programme est de réaliser le classement par moyenne d'une classe d'étudiants. Pour cela, nous devons tout d'abord définir ce qu'est un étudiant (*voir la section « La classe* `Etudiant` *»*) pour décrire ensuite une classe d'étudiants (*voir la section « La classe* `Classe` *»*). Cela fait, il devient possible de trier une classe d'étudiants selon leur moyenne (*voir la section « La méthode du tri par extraction simple »*).

La classe `Etudiant`

Un étudiant est défini par son nom (`String`), son prénom (`String`), un ensemble de notes (un tableau de `double`) et une moyenne (`double`). Ces caractéristiques constituent l'ensemble des données du type `Etudiant`.

Les comportements d'un étudiant permettent l'initialisation et l'affichage de ses caractéristiques, ainsi que le calcul de sa moyenne.

Par conséquent, nous décrivons comme suit la classe `Etudiant` :

```
public class Etudiant        {
 // Les données caractéristiques
 private String nom, prénom;
 private double [] notes, moyenne;

 // Les comportements
 public Etudiant()        {
    System.out.print("Entrer le nom de l'etudiant : ");
    nom = Lire.S();
    System.out.print("Entrer le prénom de l'etudiant : ");
    prénom = Lire.S();
    System.out.print("Combien de notes pour l'etudiant ");
    System.out.print(prénom + " " + nom + " :  ");
    int nombre = Lire.i();
    notes = new double [nombre];
    for (int i = 0; i < notes.length; i ++)    {
     System.out.print("Entrer la note  n° "+ (i + 1) + " : ");
     notes[i] = Lire.d();
    }
    moyenne = calculMoyenne();
 }
 private double calculMoyenne()      {
    double somme = 0.0;
    for(int i = 0; i < notes.length; i++) somme = somme + notes[i];
```

```
      return somme/notes.length;
   }
   public void afficheUnEtudiant()  {
      System.out.print("Les notes  de " + prénom + " " + nom + " sont : ");
      for (int i = 0; i < notes.length; i++) System.out.print("
"+notes[i]);
      System.out.println();
      System.out.println("Sa moyenne vaut " + moyenne);
   }
  public double quelleMoyenne()    {
      return moyenne;
   }
  } // Fin de class Etudiant
```

La classe `Etudiant` définit les quatre méthodes suivantes :

- `Etudiant()`. C'est le constructeur de la classe, qui permet d'initialiser l'ensemble des données de la classe `Etudiant` en demandant la saisie au clavier des nom et prénom de l'étudiant, ainsi que de l'ensemble de ses notes. Le nombre de notes peut varier d'un étudiant à un autre, puisque la valeur `nombre` est saisie en cours d'exécution.
- `calculMoyenne()`. Une fois les données saisies, le programme calcule la moyenne à l'intérieur du constructeur, grâce à la méthode `calculMoyenne()`. Cette méthode est déclarée en `private`, car, pour des raisons de sécurité, ce calcul ne peut être réalisé qu'à l'intérieur de la classe `Etudiant`.
- `afficheUnEtudiant()`. Affiche à l'écran les caractéristiques d'un étudiant.
- `quelleMoyenne()`. La donnée `moyenne` étant protégée (`private`), la méthode `quelleMoyenne()` permet l'accès en consultation, depuis l'extérieur de la classe, de la valeur mémorisée.

La classe `Classe`

Une classe d'étudiants est définie par un ensemble d'étudiants, c'est-à-dire un tableau d'objets `Etudiant`.

Les comportements d'une `classe` permettent l'initialisation, l'affichage de ses données, ainsi que le classement des étudiants dans l'ordre croissant des moyennes.

La classe `Classe` est décrite comme suit :

```
  public class Classe  {
   private Etudiant [] liste;
   public Classe()   {
      System.out.print("Nombre d'etudiants : ");
      int nbetudiants = Lire.i();
      liste = new Etudiant[nbetudiants];
      for(int i = 0; i < liste.length; i++)    liste[i] = new Etudiant();
   }
   public void afficheLesEtudiants()     {
      for (int i = 0; i < liste.length; i++) liste[i].afficheUnEtudiant();
   }
  } // Fin de class Classe
```

La donnée `liste` de la `classe` est un tableau d'objets de type `Etudiant`. Il s'agit donc là d'un tableau particulier, puisque chaque case du tableau ne correspond pas à une valeur numérique simple mais à l'ensemble des données caractéristiques d'un étudiant.

En réalité, chaque case du tableau `liste` contient l'adresse d'un objet de type `Etudiant`, comme illustré à la Figure 9-3. Cette opération est effectuée par le constructeur `Classe()`.

Ce dernier réalise la création du tableau en deux étapes. Ainsi, l'instruction :

```
liste = new Etudiant[nbetudiants];
```

crée une case mémoire `liste`, qui contient l'adresse de la première case mémoire du tableau. Ce tableau est de type `Etudiant`. Il est donc destiné à stocker les adresses des objets de type `Etudiant`.

Ensuite, la boucle :

```
for(int i = 0; i < liste.length; i++) liste[i] = new Etudiant();
```

réalise, en faisant appel au constructeur de la classe `Etudiant`, la création en mémoire des objets de type `Etudiant`, ainsi que la saisie des informations relatives à chaque étudiant.

Pour finir, chaque adresse produite par l'opérateur `new` dans la boucle `for` est placée dans chacune des cases mémoire du tableau `liste` (`liste[i]`).

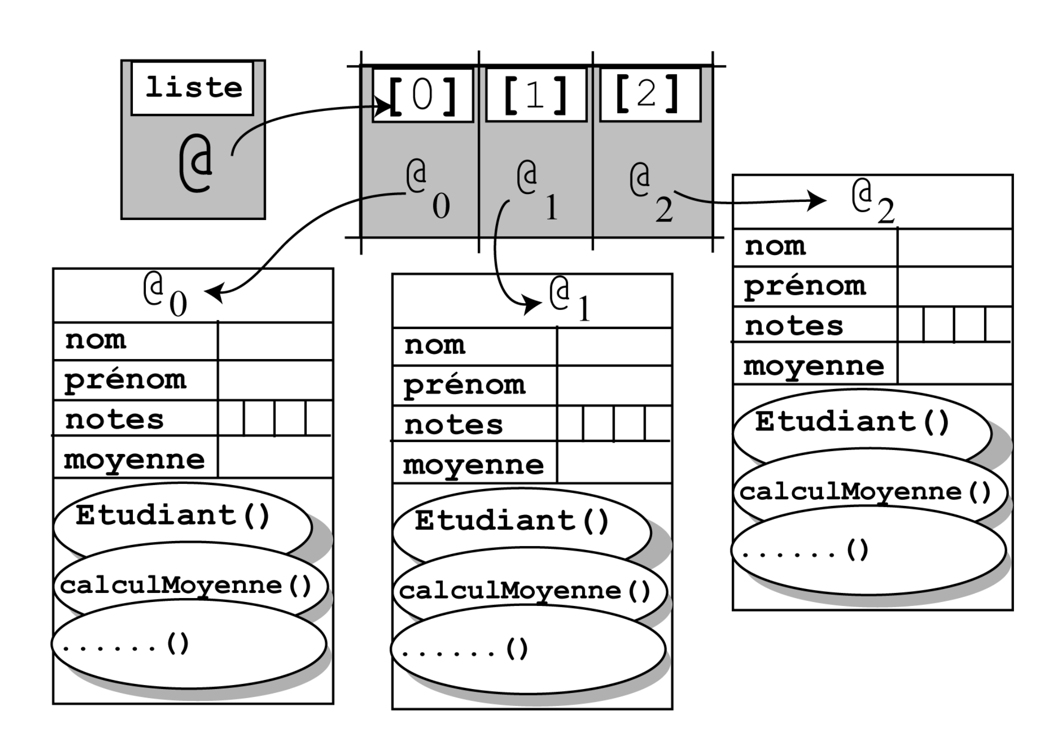


Figure 9-3.
Le tableau `Liste` *est un tableau d'objets. Chaque case du tableau mémorise l'adresse d'un objet* `Etudiant`.

La méthode `afficheLesEtudiants()` permet l'affichage des informations relatives aux étudiants en faisant appel à la méthode `afficheUnEtudiant()`, qui affiche les caractéristiques d'un étudiant à la fois.

La méthode du tri par extraction simple

Grâce aux classes `Etudiant` et `Classe`, nous sommes en mesure de créer un ensemble d'étudiants possédant chacun un nombre de notes et une moyenne. Notre objectif étant d'afficher un classement des étudiants par ordre croissant des moyennes, examinons comment trier l'ensemble des moyennes d'une classe.

L'algorithme du tri par extraction simple se décrit de la façon suivante (*voir Figure 9-4*) :

- Parcourir l'ensemble des moyennes de la classe afin de trouver la plus petite.
- Une fois trouvée, échanger cette valeur avec celle placée au tout début du tableau, de façon à être sûr que la moyenne la plus faible se trouve en début du tableau.
- Recommencer ce même traitement sur l'ensemble des moyennes moins la première, puisqu'elle vient d'être traitée.

Deux étapes sont nécessaire pour traduire cet algorithme en langage Java. Elles sont décrites ci-après, aux sections « Recherche du plus petit élément dans une liste » et « Echange de la plus petite valeur avec un élément de la liste ».

Recherche du plus petit élément dans une liste

Pour trouver la plus petite valeur d'un ensemble de valeurs, il suffit de comparer chaque valeur de la liste avec celle tout d'abord située en début de liste puis avec une plus petite, si elle existe, dans la liste. C'est ce que réalise la boucle suivante :

```
int indiceDuMin = 0 ;
for(int j = 1; j < liste.length; j++)
    if (liste[j].quelleMoyenne() < liste[indiceDuMin].quelleMoyenne())
        indiceDuMin = j;
```

Ainsi, chaque valeur du tableau (`j` variant de `1` à `liste.length`) est comparée avec la première valeur du tableau (`indiceDuMin` valant `0` en début de boucle). Si la comparaison montre que la valeur placée à l'indice `j` est plus petite que celle placée en `indiceDuMin`, alors l'indice de cette plus petite valeur est stockée dans la variable `indiceDuMin`. Le test suivant compare la valeur suivante avec la plus petite valeur qui vient d'être détectée.

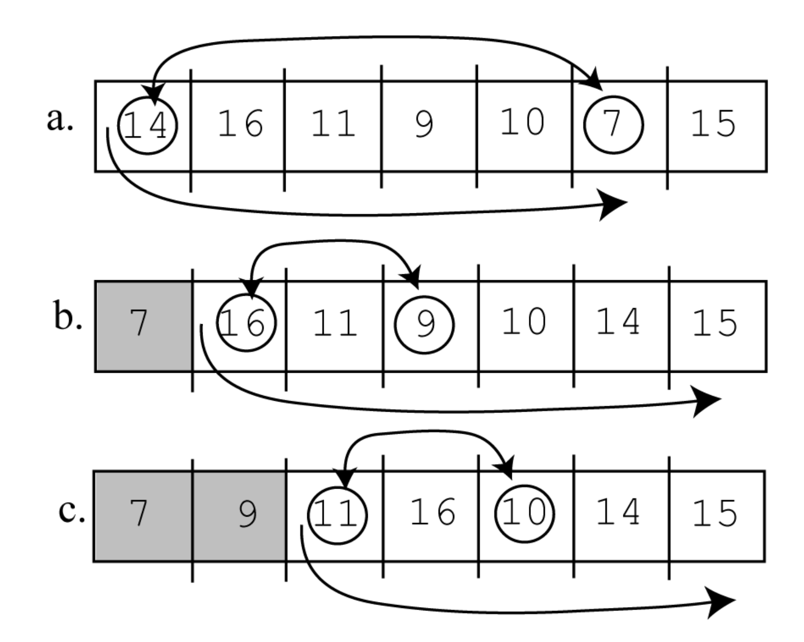


Figure 9-4.

a. *Parcours du tableau entier afin de déterminer la plus petite valeur puis échange de cette dernière avec la valeur stockée en première position dans le tableau.*
b. *Même traitement à partir de la deuxième case du tableau.*
c. *Même traitement à partir de la troisième case du tableau.*

Grâce à cette boucle, la recherche de la plus petite valeur est réalisée sur l'intégralité du tableau. Or, dans l'algorithme du tri présenté ci-dessus, cette recherche doit être réalisée dans un premier temps sur l'intégralité de la liste, puis à partir du deuxième élément, ensuite à partir du troisième élément, etc.

Cette boucle de recherche doit être placée à l'intérieur d'une méthode, de façon à pouvoir être exécutée plusieurs fois, chacune des exécutions variant en fonction d'un paramètre qui précise l'indice où débute la recherche. La méthode `ouEstLePlusPetit()`, présentée ci-dessous et à insérer dans la classe `Classe`, réalise cette recherche.

```
 private int ouEstLePlusPetit(int debut)  {
     int indiceDuMin = debut, j;
     for(j = debut+1; j < liste.length; j++)
         if (liste[j].quelleMoyenne() <
iste[indiceDuMin].quelleMoyenne())
             indiceDuMin = j;
     return indiceDuMin;
  }
```

Lorsque le programme sort de la boucle `for`, `indiceDuMin` représente l'indice de la plus petite moyenne dans le tableau `liste`. Cette valeur est alors retournée à la fonction appelante, qui l'utilise pour réaliser l'échange des valeurs.

Notez que la méthode `ouEstLePlusPetit()` est déclarée en mode `private`. Cette méthode n'est pas un comportement caractéristique d'une classe d'étudiants mais un traitement interne destiné à obtenir un classement de l'ensemble des étudiants.

Échange de la plus petite valeur avec un élément de la liste

Connaissant l'indice où se trouve la plus petite moyenne, nous devons échanger cette valeur avec celle correspondant au début de la recherche. Ce traitement est réalisé sur l'ensemble des étudiants en faisant varier l'indice du début de recherche de la première valeur du tableau jusqu'à la dernière. La méthode `classerParMoyenne()`, qui s'insère dans la classe `Classe`, réalise ces opérations :

```
 public void classerParMoyenne()  {
     int indiceDuPlusPetit ;
     Etudiant tmp;
     for(int i = 0; i < liste.length; i ++)   {
        indiceDuPlusPetit = ouEstLePlusPetit(i);
        tmp = liste[i];
        liste[i] = liste[indiceDuPlusPetit];
        liste[indiceDuPlusPetit] = tmp;
     }
  }
```

Grâce à la boucle `for`, le programme parcourt l'ensemble des étudiants de la classe. Ainsi, pour chaque étudiant de la `liste`, la boucle réalise :

- la recherche de la plus petite moyenne (`ouEstLePlusPetit()`) à partir de l'indice `i`, correspondant à l'indice de début de recherche ;

- l'échange dans la liste des données concernant l'étudiant ayant la plus petite moyenne (`indiceDuPlusPetit`) avec les données de l'étudiant placé à l'indice `i` (indice du début de recherche).

Sans revenir sur le mécanisme d'échange des données (*voir, au Chapitre 1, « Stocker une information », la section « Échanger les valeurs de deux variables »*), observez que, grâce au regroupement des données sous forme d'objets, les opérations réalisent non seulement l'échange des moyennes des étudiants, mais aussi l'ensemble des données décrivant chaque étudiant, c'est-à-dire ses nom, prénoms et notes. En effet, ce sont ici les adresses de chaque objet « `Etudiant` » qui sont échangées, et non pas simplement les moyennes.

L'application `GestionClasse`

Afin de vérifier le bon fonctionnement des classes `Etudiant` et `Classe`, il est nécessaire de construire une application qui utilise des instances de ces classes. Examinez la classe `GestionClasse`, composée d'une fonction `main()`, dans laquelle est déclaré un objet `C` de type `Classe`.

```
public class GestionClasse  {
 public static void main(String [] argument)  {
    Classe C = new Classe();
    System.out.println("-------------  Récapitulatif  -------------");
    C.afficheLesEtudiants();
    C.classerParMoyenne();
    System.out.println("-------------  Classement  ----------------");
    C.afficheLesEtudiants();
 }
} // Fin de class GestionClasse
```

En appelant le constructeur `Classe()`, le programme demande la saisie du nombre d'étudiants. Puis, pour chaque étudiant, il fait appel au constructeur `Etudiant()`, qui demande la saisie des nom, prénom et notes de l'étudiant concerné.

À la sortie du constructeur `Classe()`, le programme est en mesure d'afficher, grâce à l'instruction `C.afficheLesEtudiants()`, toutes les informations relatives à chaque étudiant de la classe `C`.

Ensuite, les étudiants sont classés par ordre croissant de moyenne grâce à l'appel de la méthode `classerParMoyenne()`, appliquée à la classe `C`. L'affichage de la liste des étudiants permet ensuite de vérifier que le tri a été correctement réalisé.

Exécution de l'application : résultats

```
Nombre d'etudiants : 4
Entrer le nom de l'etudiant : B.
Entrer le prénom de l'etudiant : Celine
Combien de notes pour l'etudiant Celine B. : 2
Entrer la note  n° 1 :  13
Entrer la note  n° 2 :  15
Entrer le nom de l'etudiant : F.
Entrer le prénom de l'etudiant : Nicolas
```

```
Combien de notes pour l'etudiant Nicolas F. :  2
Entrer la note  n° 1 :  16
Entrer la note  n° 2 :  18
Entrer le nom de l'etudiant : B.
Entrer le prénom de l'etudiant : Elodie
Combien de notes pour l'etudiant Elodie B. :  2
Entrer la note  n° 1 :  16
Entrer la note  n° 2 :  16
Entrer le nom de l'etudiant : B.
Entrer le prénom de l'etudiant : Arnaud
Combien de notes pour l'etudiant Arnaud B. :  2
Entrer la note  n° 1 :  10
Entrer la note  n° 2 :  12
------------------  Récapitulatif  ------------------
Les notes de Celine B. sont : 13.0 15.0
Sa moyenne vaut 14.0
Les notes de Nicolas F. sont : 16.0 18.0
Sa moyenne vaut 17.0
Les notes de Elodie B. sont : 16.0 16.0
Sa moyenne vaut 17.0
Les notes de Arnaud B. sont : 10.0 12.0
Sa moyenne vaut 11.0
------------------  Classement  ------------------
Les notes de Arnaud B. sont : 10.0 12.0
Sa moyenne vaut 11
Les notes de Celine B. sont : 13.0 15.0
Sa moyenne vaut 14.0
Les notes de Elodie B. sont : 16.0 16.0
Sa moyenne vaut 16.0
Les notes de Nicolas F. sont : 16.0 18.0
Sa moyenne vaut  17.0
```

Les tableaux à deux dimensions

Vous venez de voir les tableaux à une seule dimension, représentés comme une liste horizontale ou verticale d'éléments de même type. Il est possible, avec Java, de travailler aussi avec des tableaux de deux, trois, voire *n* dimensions. Pour simplifier, nous allons étudier les tableaux à deux dimensions.

Par définition, un tableau à deux dimensions s'organise non plus sur une seule ligne mais sur des lignes et des colonnes. Le croisement d'une ligne et d'une colonne détermine un élément donné du tableau.

Déclaration d'un tableau à deux dimensions

Pour déclarer un tableau à deux dimensions, la syntaxe est la suivante :

```
int [][] donnée = new int [3][5];
```

La syntaxe est pratiquement identique à la déclaration d'un tableau à une dimension. La seule différence consiste en l'ajout de `[]` supplémentaires pour signifier au compilateur que le tableau est composé de deux dimensions.

Les valeurs numériques placées entre [] derrière l'opérateur new indiquent, respectivement, le nombre de lignes puis de colonnes.

L'instruction de déclaration décrite ci-dessus réserve en mémoire un tableau nommé donnée, composé de 3 lignes et de 5 colonnes. Chaque élément du tableau étant un entier, l'opérateur new réserve 3 * 5, soit 15 cases mémoire de la taille d'un entier.

Remarquez que le nombre de lignes d'un tableau est donné par l'expression donnée.length, alors que le nombre de colonnes est déterminé par l'expression donnée[0].length. En effet, le nombre de colonnes d'un tableau correspond au nombre d'éléments placés sur une ligne (*voir Figure 9-5*).

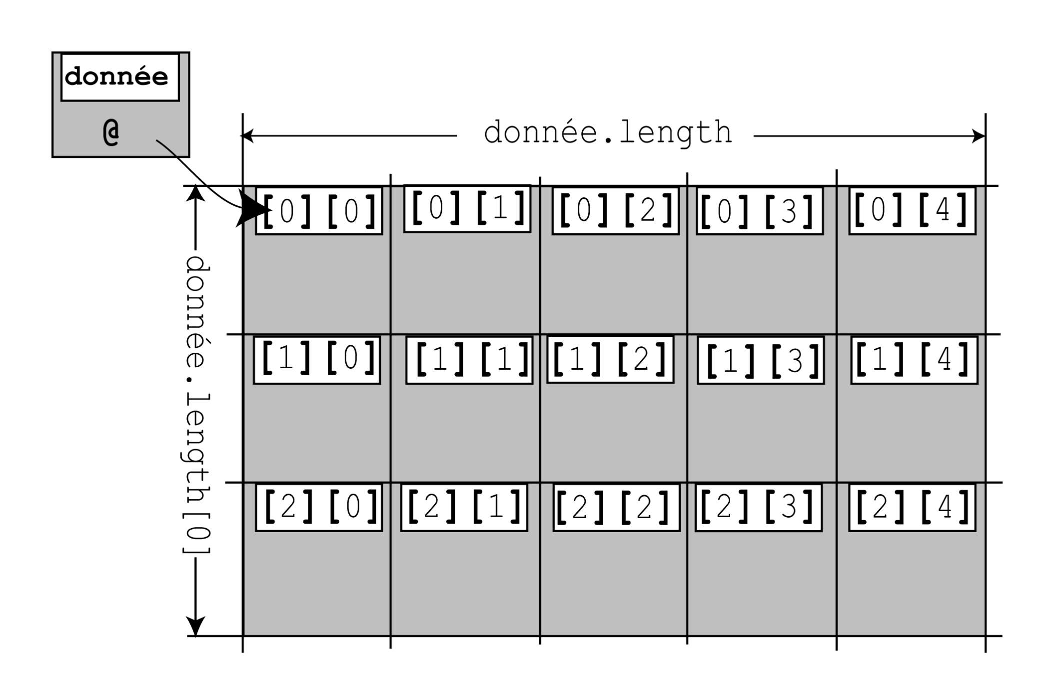


Figure 9-5.
Un tableau s'organise sur des lignes et des colonnes numérotées à partir de [0][0].

Accéder aux éléments d'un tableau

Pour initialiser, modifier ou consulter la valeur d'un élément d'un tableau, il convient d'utiliser deux indices : un indice pour les lignes et un indice pour les colonnes. Chaque indice étant contrôlé par une boucle for, la technique consiste à imbriquer deux boucles de la façon suivante :

```
for (int i = 0; i < donnée.length; i++)
     for (int j = 0; j < donnée[0].length; j++)
     // donnée[i][j]) = uneValeur ;
```

La boucle j est imbriquée dans la boucle i. Les variables i et j sont les compteurs de boucles qui contrôlent respectivement les lignes et colonnes du tableau donnée.

Pour mieux comprendre les mécanismes de manipulation des tableaux et, en particulier, le déroulement des valeurs des indices à l'intérieur des boucles for, examinons l'exemple suivant.

Exemple : Dessiner un sapin

Les tableaux à deux dimensions sont très souvent utilisés pour stocker les images. En effet, une image affichée à l'écran correspond en réalité à une surface découpée en lignes

et colonnes. La donnée numérique se situant à la croisée de ces lignes et colonnes représente un point de l'image et a pour valeur la couleur d'affichage à l'écran.

Cahier des charges

L'objectif de cet exemple est de dessiner à l'écran un sapin de Noël décoré, comme l'illustre la figure suivante :

```
    %
   ...
  .....
 .%.%...
........%
%.%......%.
```

Pour simplifier à l'extrême la lisibilité du programme, nous n'utilisons que de simples caractères alphanumériques pour afficher notre sapin. C'est pourquoi son affichage reste assez sommaire. Pour voir de plus « jolis » sapins, reportez-vous au Chapitre 11, « Dessiner des objets ».

Créer et afficher un triangle composé de trois lignes

Fidèles au principe de décomposition d'un problème, nous allons chercher dans un premier temps à afficher la forme suivante :

```
  .
 ...
.....
```

L'affichage de cette forme correspond à un triangle. Sa structure interne est définie en mémoire à l'aide d'un tableau à deux dimensions. Il s'agit d'un tableau composé de 3 lignes et de 5 colonnes, comme l'illustre le tableau suivant :

```
00100
01110
11111
```

Ce tableau est constitué de valeurs numériques placées de telle façon que le programme dessine un triangle en affichant un point lorsque la valeur du tableau vaut 1 et sinon une espace.

Pour réaliser astucieusement l'initialisation de ce tableau, examinons l'emplacement des valeurs par rapport aux indices du tableau. Sachant qu'un tableau est toujours initialisé à `0` lors de sa création en mémoire par l'opérateur `new`, observez uniquement les indices correspondant aux valeurs égales à `1`.

Ligne\Colonne	[0]	[1]	[2]	[3]	[4]
[0]	0	0	1	0	0
[1]	0	1	1	1	0
[2]	1	1	1	1	1

Remarquez, à la colonne [2], que toutes les valeurs sont initialisées à 1. Cette colonne correspond en réalité à la colonne du milieu du tableau. En supposant que l'ensemble des valeurs soit stocké dans un tableau nommée sapin, l'indice de cette colonne est obtenu grâce à l'instruction

```
int [][] sapin = new int [3][5] ;
int milieu = sapin[0].length / 2;
```

L'expression sapin[0].length correspondant au nombre de colonnes, soit 5, la variable milieu prend pour valeur 5/2, soit 2 en entier.

Ensuite, les valeurs situées de part et d'autre de cette colonne sont, elles aussi, initialisées à 1. Pour la ligne numéro [1], seul **un** élément, à droite et à gauche du milieu, est initialisé à 1. Pour la ligne numéro [2], **deux** éléments, à droite et à gauche, valent 1.

Il y a donc corrélation entre le nombre de valeurs à initialiser et le numéro de la ligne sur laquelle l'initialisation est effectuée. C'est pourquoi le traitement se réalise de la façon suivante :

```
for ( int i = 0 ; i < sapin.length ; i++)    {
    for ( int j = -i; j <= i; j++)     {
        sapin[i][milieu+j] = 1;
    }
}
```

La variable i partant de 0 jusqu'à sapin.length (soit 3) examine toutes les lignes du tableau. Pour chaque ligne, grâce à la seconde boucle en j, les valeurs du tableau sont initialisées à 1, de part et d'autre du milieu. Pour mieux comprendre le déroulement des opérations, examinez le tableau d'évolution des variables.

i	j	milieu	tab[i][milieu+j]
0	0	2	tab[0][0 + 2] = 1
0	**1**	2	// sortie de boucle
1	-1	2	tab[1][-1 + 2] = 1
1	0	2	tab[1][0 + 2] = 1
1	1	2	tab[1][1 + 2] = 1
1	**2**	2	// sortie de boucle
2	-2	2	tab[2][-2 + 2] = 1
2	-1	2	tab[2][-1 + 2] = 1
2	0	2	tab[2][0 + 2] = 1
2	1	2	tab[2][1 + 2] = 1
2	2	2	tab[2][2 + 2] = 1
2	**3**	2	// sortie de boucle
3		2	// sortie de boucle

Une fois le tableau créé et initialisé en mémoire, l'affichage du dessin s'effectue en testant la valeur de chaque point du tableau. Si la valeur est nulle, une espace est affichée, sinon, un point est affiché. Traduite en Java, cette marche à suivre s'écrit à l'aide de deux boucles imbriquées, comme suit :

```
for (int i = 0; i < sapin.length; i++) {
    for (int j = 0; j < sapin[0].length; j++) {
         if(sapin[i][j] == 0) {
            System.out.print(" ");
        else
            System.out.print(".");
    }
    System.out.println();
}
```

L'indice `i` représente les lignes, tandis que `j` représente les colonnes. Grâce aux boucles imbriquées, chaque intersection des lignes et colonnes est consultée, de façon à afficher le caractère correspondant à la valeur stockée. Lorsque la boucle `i` est terminée, cela signifie que tous les éléments (colonnes) de la ligne `i` ont été affichés, et il est nécessaire de passer à la ligne suivante de l'écran, grâce à l'instruction `System.out.println()`.

Créer un triangle composé de `n` lignes

Nous avons créé un triangle à 3 lignes composé de 5 colonnes. Si nous souhaitons ajouter une nouvelle ligne, nous devons obligatoirement ajouter deux colonnes supplémentaires. La relation entre le nombre de lignes et de colonnes s'exprime selon l'équation :

```
Nombre de colonnes = 2 * Nombre de lignes - 1
```

Pour un triangle possédant `3` lignes, vous obtenez `2 * 3 - 1 = 5` colonnes. Pour un triangle composé de `4` lignes, vous obtenez `2 * 4 - 1 = 7` colonnes. L'équation reste valide pour un triangle à une ligne, puisque le nombre de colonnes vaut `2 * 1 - 1 = 1`.

Les instructions suivantes permettent de créer un triangle dont le nombre de lignes est déterminé par l'utilisateur :

```
public class Sapin {
 public static void main(String  [] arg)  {
  System.out.print("Nombre de ligne :  ");
  int nl = Lire.i();
  if (nl <= 0) {
   System.out.println("Le nombre de lignes doit être supérieur a 0 ");
   System.exit(0);
  }
  int nc = 2*nl-1;
    int [][] sapin = new int[nl][nc];
    int milieu = sapin[0].length/2;
    for ( int i = 0 ; i < nl ; i++)     {
     for ( int j = -i; j <= i; j++)     {
         sapin[i][milieu+j] = +1;
     }
```

```
      }
   } // Fin de la fonction main()
 } // Fin de la classe Sapin
```

Placer des décorations au hasard

Cela fait, nous sommes en mesure d'afficher le sapin sans décoration puisque, pour chaque élément du tableau valant 1, un point est affiché. Pour ajouter quelques décorations, l'idée est de placer au hasard d'autres valeurs que 1. Ainsi, l'affichage peut être modulé en fonction de la valeur rencontrée.

Pour placer dans notre sapin de nouvelles valeurs au hasard, comprises entre `1` et `6`, par exemple, il suffit de modifier l'initialisation du tableau de la façon suivante :

```
for ( int i = 0 ; i < nl ; i++)
for ( int j = -i; j <= i; j++)
   sapin[i][milieu + j] = (int ) (5 * Math.random() + 1);
```

L'affichage du sapin se déroule ensuite comme suit :

```
    for (int i = 0; i < sapin.length; i++) {
         for (int j = 0; j < sapin[0].length; j++) {
          switch (sapin[i][j]) {
             case 0 : System.out.print(" ");
             break;
             case 2 : System.out.print("%");
             break;
             default : System.out.print(".");
          }
         }
         System.out.println();
}
```

Suivant la valeur contenue en `sapin[i][j]`, le programme affiche une espace, un point ou un `%`. Remarquez que les valeurs `1`, `3`, `4`, `5` et `6` affichent toutes un point. Seule la valeur `2` permet l'affichage d'une guirlande. Ce choix a pour effet d'afficher volontairement plus de points que de guirlandes de façon à obtenir un sapin qui ne soit pas trop surchargé.

Attention aux boucles imbriquées

Pour manipuler des tableaux à deux dimensions, le programmeur utilise deux boucles `for` imbriquées. Dans ce cas, une boucle `for` est placée à l'intérieur d'une première boucle `for`. Ce type d'écriture nécessite attention, car certaines erreurs peuvent empêcher le bon déroulement du programme.

Une erreur d'inattention commise, en particulier, à cause des facilités du copier-coller peut aboutir à programmer deux boucles imbriquées qui utilisent la même variable comme compteur de boucles.

Ainsi, en écrivant :

Boucles imbriquées (`i` est compteur de boucles)	Variation de la variable `i`

`int i;`
`for(i = 1; i <= 3; i= i+1) {`
`// Boucle 1`
`    for(i = 1; i <= 4; i= i+1) {`
`    // Boucle 2`
`    }`
`}`
	`Boucle 1 : i = 1, i <= 3`
	`    Boucle 2 : i = 1, i <= 4`
	`    Boucle 2 : i = 2, i <= 4`
	`    Boucle 2 : i = 3, i <= 4`
	`    Boucle 2 : i = 4, i <= 4`
	`    Boucle 2 : i = 5, i > 4`
	`Boucle 1 : i = 6, i > 3`

Le compteur de boucles `i` est déclaré à l'extérieur des boucles. Les deux boucles utilisent la même case mémoire pour stocker les variations de la valeur de `i`.

En entrant dans la `Boucle 1`, `i` prend la valeur `1`, de même qu'en entrant dans la `Boucle 2`. Puis `i` est incrémenté de `1` à chaque tour de la `Boucle 2`, jusqu'à ce que `i` dépasse la valeur `4`. La `Boucle 2` est alors terminée. `i` vaut par conséquent `5`. On entre à nouveau dans la `Boucle 1`. `i` est incrémenté de `1` (`i` vaut `6`) puis testé. `6` étant supérieur à `3`, la `Boucle 1` est terminée. La `Boucle 1` n'est donc parcourue qu'une seule fois au lieu de trois.

Remarquez que :

- Si le compteur de boucles est déclarée à l'intérieur de la boucle, comme suit :

```
for(int i = 1; i <= 3; i= i+1) {
// Boucle 1
    for( int i = 1; i <= 4; i= i+1) {
    // Boucle 2
    }
}
```

  alors, le compilateur détecte une erreur du type : `Variable 'i' is already defined in this method`. En effet, le fait de déclarer un compteur de boucles portant le même nom, dans chaque boucle, revient à déclarer, dans un même bloc de programme, deux variables portant le même nom.

- L'écriture de deux boucles non imbriquées utilisant la même variable comme compteur de boucles n'est pas une erreur.

```
for( int i = 1; i <= 3; i= i+1) {
    // Premier for, 4 tours
}
for( int i = 1; i <= 6; i= i+1) {
    // Deuxième for, 7 tours
}
```

  L'emploi d'une même variable de compteur pour deux boucles disjointes est correct. En effet, la variable `i` est déclarée dans la boucle. Elle n'existe en mémoire que le temps d'utilisation de la boucle. Lorsque `i` est à nouveau déclarée dans la deuxième boucle, la variable `i` précédente n'existe déjà plus. Cette manière de programmer est courante, car elle facilite la lecture des boucles. Très souvent, les compteurs de boucles ont pour nom `i`, `j` ou `k`.

Résumé

Les tableaux sont utilisés pour regrouper sous un même nom de variable un nombre donné de valeurs de même type. Un tableau est défini par :

- un nom ;
- un type ;
- le nombre de dimensions ;
- une taille pour chacune des dimensions.

Déclaration d’un tableau

Comme toute variable, un tableau doit être déclaré. La syntaxe est la suivante :

- Pour un tableau à **une dimension** :

  ```
  float [] donnee    = new float [5] ;
  ```

  Cette instruction déclare un tableau composé de nombres réels de simple précision, appelé `donnee`. L’opérateur `new` réserve 5 cases mémoire de 4 octets chacune.

- Pour un tableau à **deux dimensions** :

  ```
  int [][] valeur    = new int [3][2] ;
  ```

  Cette instruction déclare un tableau à deux dimensions, composé de nombres entiers, appelé `valeur`. L’opérateur `new` réserve `3 * 2 = 6` cases mémoire de `4` octets chacune.

La **taille** d’un tableau est une valeur entière définie soit à l’intérieur du programme, soit saisie au clavier lors de l’exécution du programme. Une fois la taille fixée par l’opérateur `new`, il n’est plus possible de la modifier en cours d’exécution.

Un tableau n’est pas nécessairement de type simple (`int`, `double`, etc.). Il peut être de type structuré (`String` ou type défini par le programmeur, etc.). Dans ce cas, le tableau est un tableau d’objets stockant dans chacune de ses cases l’adresse d’un objet à mémoriser.

Pour accéder à une case (élément) du tableau, il suffit de placer, derrière le nom du tableau, le numéro de la case (**indice**) entre `[]`. Chaque indice est une expression entière. La première valeur d’un tableau est stockée à l’indice `0` du tableau et non à l’indice `1`.

```
for (int i = 0; i < donnee.length; i++)
  System.out.println(" " + donnee[i]);
```

Ainsi, la boucle `for` ci-dessus permet d’accéder à chaque élément du tableau. Pour ne pas dépasser la taille du tableau, il est conseillé d’utiliser la donnée `length`, qui correspond à la longueur du tableau. Le programme dépasse la taille du tableau lorsque la valeur de l’indice est supérieure à la taille déclarée du tableau. L’interpréteur détecte alors une erreur du type : `java.lang.ArrayIndexOutOfBoundsException`.

Exercices

Les tableaux à une dimension

9.1 Qu'affiche le programme suivant ?

```
int i ;
int []     valeur = new int[6] ;
valeur [0] = 1;

for (i = 1; i < valeur.length; i++)
 valeur[i] = valeur[i-1]+2;
for (i = 0; i < valeur.length; i++)
  System.out.print("valeur["+i+"] = " + valeur[i]);
```

9.2 Écrivez un programme qui :

a. Stocke dans un tableau des valeurs entières passées en paramètres de la ligne de commande.

b. Calcule la somme de ces valeurs.

c. Calcule la moyenne de ces valeurs.

d. Recherche la plus grande valeur du tableau.

e. Détermine la position de la plus grande valeur.

f. Affiche le nombre de valeurs supérieures à la moyenne.

Les tableaux d'objets

9.3 En reprenant la classe `Cercle`, définie au Chapitre 8, « Les principes du concept d'objet », écrivez un programme qui :

a. Crée un tableau de type Cercle, dont la taille soit choisie par l'utilisateur. Si le nombre de cercles créés est inférieur à 4, le programme initialise par défaut la taille du tableau à 4.

b. Initialise les données de chaque tableau à l'aide du constructeur par défaut de la classe `Cercle`.

c. Déplace le cercle n° 1 en 20, 20.

d. Agrandit le cercle n° 2 de 50.

e. Échange le cercle n° 0 avec le n° 3.

f. Permute les cercles, de façon que le cercle 0 soit stocké en 1, le cercle 1 en 2, … et le cercle 3 en 0.

Les tableaux à deux dimensions

9.4 Écrivez un programme qui :

a. À l'aide de boucles imbriquées, initialise la matrice 7 * 7 aux valeurs suivantes :

```
1  0  0  1  0  0  1
0  1  0  1  0  1  0
0  0  1  1  1  0  0
1  1  1  1  1  1  1
0  0  1  1  1  0  0
0  1  0  1  0  1  0
1  0  0  1  0  0  1
```

✔ Les compteurs de boucles seront astucieusement choisis afin d'initialiser automatiquement le tableau.

b. Affiche à l'écran le tableau, en remplaçant les valeurs :

0 par un espace (" ") ;

1 par un astérisque ("*").

Pour mieux comprendre le mécanisme des boucles imbriquées `for-for`

9.5 Afin d'exécuter le programme suivant :

```
public class Exercice5 {
 public static void main (String [] parametre) {
  int i,j, N = 5;
  char C;
  System.out.print("Entrer un  caractère :");
  C = Lire.c();
  for (i = 1; i < N; i++)  {
   for (j = 1; j < N; j++)  {
    if (i < j) System.out.print(C);
    else System.out.print(" ");
   }
  }
 }
}
```

a. Examinez le code source, repérez les instructions concernées par les deux boucles répétitives, et déterminez les instructions de début et de fin de boucle.

b. Quelles sont les instructions qui permettent de modifier le résultat du test de sortie de boucle ?

c. En supposant que l'utilisateur entre la valeur « ! », exécutez le programme suivant à la main (pour vous aider, construisez le tableau d'évolution de chaque variable déclarée).

d. Quel est le résultat affiché à l'écran ?

9.6 En construisant le tableau d'évolution de la variable `i`, que constatez-vous lors de l'exécution de ces boucles ?

```
for(i = 1; i <= 5; i = i+1)
{
    for(i = 1; i <= 2; i= i+1)
    {
        System.out.print("i = "+i);
    }
}
```

Le projet « Gestion d'un compte bancaire »

Traiter dix lignes comptables

L'objectif est de traiter, non plus une seule ligne comptable, mais dix lignes comptables. Pour cela, vous devez, dans un premier temps, modifier la déclaration de la donnée `ligne`, dans la classe `Compte`, comme suit :

```
private LigneComptable [] ligne;
```

Comme le nombre de lignes comptables est fixé dans le cahier des charges, il est possible de définir une constante comme suit :

```
public static final int NBLigne = 10 ;
```

`NBLigne` représente le nombre maximal de lignes comptables à traiter. Les lignes comptables étant créées au fur et à mesure des opérations réalisées par l'utilisateur, il est nécessaire de définir une variable (`nbLigneRéel`), qui compte le nombre de lignes comptables effectivement créées en cours d'exécution du programme.

La gestion des lignes comptables entraîne la modification des méthodes `Compte()`, `créerLigne()` et `afficherCompte()`.

Transformer les constructeurs `Compte()`

Dans chaque constructeur :

a. À l'aide de l'opérateur `new`, créer en mémoire la donnée `ligne`, sous forme d'un tableau de dix lignes comptables.

b. Initialiser la variable `nbLigneRéel` à `–1`, puisqu'aucune ligne n'a encore été saisie.

Transformer la méthode `créerLigne()`

Lorsque le nombre de lignes comptables traité est supérieur à 10, le programme doit effacer la première ligne traitée, de façon à décaler les suivantes (la deuxième allant en première position, la troisième en deuxième position, etc.) afin de pouvoir stocker la nouvelle ligne en dernière position du tableau `ligne`.

La méthode `créerLigne()` réalise ce traitement de la façon suivante :

a. Incrémente `nbLigneRéel` de `1`.

b. Si le nombre de lignes créées est inférieur à `NBLigne`, crée en mémoire une ligne comptable grâce au constructeur de la classe `LigneComptable` et stocke en mémoire son adresse dans le tableau `ligne`.

c. Si le nombre de lignes est supérieur à `NBLigne`, décale toutes les lignes vers le haut, grâce à la méthode `décalerLesLignes()` décrite ci-dessous, et stocke la nouvelle ligne comptable en dernière position (`NBLigne - 1`) du tableau `ligne`.

```
private void décalerLesLignes() {
 for(int i = 1; i < NBLigne ; i++)
         ligne[i-1] = ligne[i];
}
```

d. Modifie la valeur courante du compte en fonction du crédit ou débit réalisé par la nouvelle ligne comptable.

Transformer la méthode `afficherCompte()`

Modifier la méthode `afficherCompte()` de façon à afficher l'ensemble des lignes saisies en cours d'exécution du programme.

10

Collectionner un nombre indéterminé d’objets

Comme nous l’avons vu au cours du chapitre précédent, les tableaux permettent la manipulation rapide et efficace d’un ensemble de données. Cependant, leur principal inconvénient est d’être de taille fixe. Ainsi, l’ajout d’un élément dans un tableau demande une gestion rigoureuse des indices afin d’éviter que ces derniers ne prennent une valeur supérieure à la taille du tableau.

Pour pallier cette difficulté majeure pour un grand nombre de programmes, le langage Java propose plusieurs outils de manipulation des données en mémoire vive, au fur et à mesure des besoins de l’application. Ces outils sont présentés et analysés à la section « La programmation dynamique ».

En outre, lorsqu’un programme utilise des collections importantes de données, il doit les archiver de façon à ne pas les voir disparaître après l’arrêt de l’application ou de l’ordinateur. Le langage Java offre différentes méthodes pour réaliser ce stockage de données. Elles sont étudiées à la section « L’archivage de données ».

La programmation dynamique

À la différence de la programmation **statique**, dans laquelle le nombre de données géré par l’application est fixé une fois pour toutes lors de l’exécution du programme, la programmation **dynamique** offre l’avantage de gérer un nombre indéterminé d’objets, en réservant des espaces mémoire, au fur et à mesure des besoins de l’utilisateur.

Cette technique se montre très utile lorsque le nombre d’objets à traiter n’est pas connu ni définissable avant l’exécution du programme. Par exemple, tous les logiciels de gestion, et c’est une grande part des programmes informatiques, se doivent de gérer les données qu’ils traitent de façon dynamique.

En effet, sans programmation dynamique, vous pourriez voir une bibliothèque refuser de nouveaux lecteurs sous prétexte que le logiciel qu'elle utilise ne serait pas en mesure de traiter plus de 50 000 inscriptions, ou encore voir un logiciel de traitement de texte s'interrompre parce qu'il lui serait impossible de gérer la saisie et l'affichage de plus 10 000 caractères.

Pour éviter de telles situations, le langage Java propose différents outils qui gèrent dynamiquement les données d'un programme. En particulier, il existe des objets de type `Vector`, dont nous analysons les caractéristiques à la section « Les vecteurs ». Les objets de type `Hashtable`, étudiés à la section « Les dictionnaires », offrent aussi l'avantage de gérer les données de façon dynamique, tout en organisant l'information de façon à faciliter son exploitation.

Les vecteurs

Les vecteurs sont des objets de type `Vector`, un type prédéfini du langage Java. La gestion des vecteurs est assez similaire à la gestion d'un tableau puisque le programme crée une liste par ajout de données au fur et à mesure des besoins de l'utilisateur. Les données sont enregistrées dans leur ordre d'arrivée. Un indice géré par l'interpréteur permet de retrouver l'information.

Manipulation d'un vecteur

Pour utiliser un vecteur, il est nécessaire de le déclarer de la façon suivante :

```
Vector liste = new Vector() ;
```

Ainsi déclaré, `liste` est un objet de type `Vector`, auquel on peut appliquer des méthodes de la classe `Vector`. Ces méthodes, décrites au tableau ci-dessous, permettent l'ajout, la suppression ou la modification d'une donnée dans le vecteur (`liste` par exemple).

Opération	Fonction Java
Ajoute un élément `objet` en fin de liste.	`add(objet)`
Insère un élément `objet` dans la liste, à l'`indice` spécifié en paramètre.	`add(indice, objet)`
Ajoute un élément `objet` en fin de liste et augmente sa taille de un.	`addElement(objet)`
Retourne l'élément stocké à l'`indice` spécifié en paramètre.	`elementAt(indice)`
Supprime tous les éléments de la liste.	`clear()`
Retourne l'indice dans la liste du premier `objet` donné en paramètre, ou `-1` si `objet` n'existe pas dans la liste.	`indexOf(objet)`
Retourne l'indice dans la liste du dernier `objet` donné en paramètre, ou `-1` si `objet` n'existe pas dans la liste.	`lastIndexOf()`
Supprime l'`objet` dont l'`indice` est spécifié en paramètre.	`remove(indice)`
Supprime tous les éléments compris entre les indices `i` (valeur comprise) et `j` (valeur non comprise).	`removeRange(i, j)`
Remplace l'élément situé en position `i` par l'`objet` spécifié en paramètre.	`setElementAt(objet, i)`
Retourne le nombre d'éléments placés dans la liste.	`size()`

Exemple : Créer un nombre indéterminé d'étudiants

Pour mieux comprendre l'utilisation des vecteurs, reprenons l'exemple de la classe d'étudiants (*voir, au chapitre précédent, « Collectionner un nombre fixe d'objets », la section « Trier un ensemble de données »*), de façon que le programme traite, non plus un nombre fixe d'étudiants, mais un nombre indéterminé.

La classe `Etudiant` n'est pas à modifier, puisque l'objectif est de transformer uniquement la gestion en mémoire des étudiants. La correction se porte donc sur la classe `Classe`, où la `liste` d'étudiants doit être déclarée de type `Vector` au lieu d'être déclarée sous forme de tableau. Examinons la nouvelle classe `Classe` :

```
import java.util.*;
 public class Classe {
 private Vector liste;
 public Classe()   {
    liste = new Vector();
 }
 public void ajouteUnEtudiant() {
    liste.addElement(new Etudiant());
 }
public void afficheLesEtudiants()     {
    int nbEtudiants = liste.size();
    if (nbEtudiants > 0) {
        Etudiant tmp;
        for (int i = 0; i < nbEtudiants; i ++) {
            tmp = (Etudiant) liste.elementAt(i);
            tmp.afficheUnEtudiant();
        }
    }
    else System.out.println("Il n'y a pas d'etudiant dans cette liste");
 }
} // Fin de Classe
```

Les outils comme `Vector` sont proposés par le langage Java. Ils sont définis à l'intérieur de classes, qui ne sont pas, par défaut, directement accessibles par le compilateur. C'est pourquoi le programmeur doit préciser au compilateur où se situe la librairie du langage Java définissant l'outil utilisé (`package`). Pour ce faire, il doit placer une instruction `import` en première ligne du fichier qui utilise l'outil souhaité.

La classe `Vector` étant définie dans le package `java.util`, il convient de placer l'instruction `import.java.util.*;` en tête du fichier. En effet, si cette instruction fait défaut, le compilateur détecte une erreur du type : `Class Vector not found`.

Cela fait, la donnée `liste`, déclarée de type `Vector`, doit être manipulée en tant que telle dans chaque méthode de la classe. Les trois méthodes suivantes sont définies à l'intérieur de celle-ci :

- Le constructeur `Classe()`, qui fait appel au constructeur de la classe `Vector` afin de déterminer l'adresse du premier élément de la `liste`.
- La méthode `ajouteUnEtudiant()`, qui place un élément dans la `liste` grâce à la méthode `addElement()`. L'élément ajouté à la liste est un objet de type `Etudiant`, créé par l'intermédiaire du constructeur `Etudiant()`, qui demande la saisie au clavier des

données caractéristiques de l'étudiant à construire. Cela fait, la taille de la liste est automatiquement augmentée de `1` par l'interpérteur.

Observez que l'ajout d'un élément dans un vecteur n'est possible que si l'élément est un objet. Il n'est, en effet, pas possible d'ajouter une valeur de type simple, telle que `int`, `float` ou `double`. Pour réaliser une telle opération, il est nécessaire de transformer les types simples en leur homologue structuré. Par exemple, le type simple `int` peut être représenté par le type `Integer`, définissant un objet contenant une valeur numérique entière (*pour plus de précisions, voir l'exercice 1, en fin de chapitre*).

- La méthode `afficheLesEtudiants()` parcourt l'ensemble de la liste grâce à la méthode `elementAt()`, qui fournit en résultat l'élément stocké à la position spécifiée en paramètre, soit `i`. Ce résultat, pour être consultable, doit obligatoirement être « casté » en `Etudiant` (*voir, au Chapitre 1, « Stocker une information », la section « La transformation de types »*). En effet, un vecteur a la capacité de mémoriser n'importe quel type d'objet, prédéfini ou non. Il est donc nécessaire d'indiquer au compilateur à quel type correspond l'objet extrait.

  L'indice `i`, variant de `0` jusqu'à la taille effective (`nbEtudiants`) de la liste, l'ensemble des étudiants contenus dans la `liste` est affiché.

Exemple : L'application `GestionClasse`

Observons l'application `GestionClasse`, qui définit et utilise un objet `Classe`.

```
public class GestionClasse  {
    public static void main(String [] argument)  {
    byte choix = 0 ;
    Classe C = new Classe();
    do   {
        System.out.println("1. Ajoute un etudiant");
        System.out.println("2. Affiche la classe");
        System.out.println("3. Pour sortir");
        System.out.print("Votre choix : ");
        choix  = Lire.b();
        switch (choix) {
            case 1 :    C.ajouteUnEtudiant();
                        break;
            case 2 :    C.afficheLesEtudiants();
                        break;
            case 3 :    System.exit(0);
           default :   System.out.println("Cette option n'existe pas ");
        }
    } while (choix != 3);
 }
}
```

Le nombre d'étudiants à traiter n'est pas déterminé à l'avance. C'est pourquoi l'application `GestionClasse` réalise, grâce à la mise en place d'une boucle `do…while`, la saisie des données au fur et à mesure des besoins de l'utilisateur. Le programme laisse le `choix` à l'utilisateur de saisir de nouveaux étudiants ou d'afficher ceux effectivement saisis. L'exécution de cette application a pour résultat à l'écran :

```
1. Ajoute un etudiant
2. Affiche la classe
3. Pour sortir
Votre choix : 2
Il n'y a pas d'etudiant dans cette liste
1. Ajoute un etudiant
2. Affiche la classe
3. Pour sortir
Votre choix : 1
Entrer le nom de l'etudiant : M.
Entrer le prenom de l'etudiant : Sandra
Combien de notes pour l'etudiant  Sandra M. :  2
Entrer la note  n° 1 :  15
Entrer la note  n° 2 :  13
1. Ajoute un etudiant
2. Affiche la classe
3. Pour sortir
Votre choix : 1
Entrer le nom de l'etudiant : Philippe
Entrer le prenom de l'etudiant : T.
Combien de notes pour l'etudiant  Philippe T. :  2
Entrer la note  n° 1 :  12
Entrer la note  n° 2 :  8
1. Ajoute un etudiant
2. Affiche la classe
3. Pour sortir
Votre choix : 2
Les notes de Sandra M. sont :  15.0 13.0
Sa moyenne vaut  14.0
Les notes de Philippe T. sont :  12.0 8.0
Sa moyenne vaut  10.0
1. Ajoute un etudiant
2. Affiche la classe
3. Pour sortir
Votre choix : 3
```

L'utilisation d'objets du type `Vector` est souple et facilite amplement la vie du programmeur lorsque ce dernier souhaite écrire une application qui gère des données de façon dynamique. Les méthodes de la classe `Vector` permettent aussi la recherche ou l'insertion de nouveaux éléments grâce, en particulier, à la méthode `indexOf(objet)`, qui retrouve l'indice de l'objet spécifié en paramètre dans la liste.

Cependant, lorsque la liste mémorise des objets complexes, tels que les données caractéristiques d'un étudiant, la recherche d'un étudiant particulier n'est pas simple. En effet, il est nécessaire de fournir au programme toutes les données de l'étudiant (nom, prénom, notes et moyenne), de façon à être sûr de le retrouver dans la liste. La méthode `indexOf(objet)` ne retrouve l'objet spécifié en paramètre qu'à la seule condition qu'il soit totalement identique à celui stocké dans la `liste`.

Les dictionnaires

Pour améliorer la recherche d'éléments complexes dans une liste, la technique consiste à organiser les données, non plus par rapport à un indice, mais par rapport à une clé explicite. De cette façon, la recherche d'un objet dans la liste s'effectue, non plus sur l'ensemble des données qui le composent, mais sur une clé unique qui lui est associée.

L'organisation de données, par association d'une clé à un ensemble de données, est appelée un dictionnaire. Dans un dictionnaire, chaque définition est associée au mot qu'elle définit et non pas à sa position (numérique) dans le dictionnaire.

Manipulation d'un dictionnaire

Le type `Hashtable` proposé par le langage Java permet de réaliser simplement l'association clé-élément. Les méthodes associées à ce type sont la création, la suppression, la consultation ou la modification d'une association.

Pour utiliser un dictionnaire, il est nécessaire de le déclarer de la façon suivante :

```
Hashtable listeClassée = new Hashtable () ;
```

Ainsi déclaré, `listeClassée` est un objet de type `Hashtable`, sur lequel on peut appliquer des méthodes de la classe `Hashtable`. Les méthodes les plus couramment utilisées sont décrites au tableau ci-après.

Opération	Fonction Java
Place dans le dictionnaire l'association `clé-objet`.	`put(clé, objet)`
Retourne l'objet associé à la `clé` spécifiée en paramètre.	`get(clé)`
Supprime dans le dictionnaire l'association `clé-objet` à partir de la `clé` spécifiée en paramètre.	`remove(clé)`
Retourne le nombre d'associations définies dans le dictionnaire.	`size()`

Exemple : Créer un dictionnaire d'étudiants

Pour mieux comprendre l'utilisation de tels objets, modifions le programme de gestion d'une classe d'étudiants de façon à organiser les données à partir d'une clé définie par le programme.

Définir une clé d'association

En supposant qu'un étudiant soit totalement identifiable par son nom et son prénom, la clé d'association des données est définie comme étant une chaîne de caractères majuscules, dont le premier caractère coïncide avec le premier caractère du prénom de l'étudiant et dont les caractères suivants correspondent au nom de l'étudiant.

De cette façon, chaque clé est déterminée par programme, indépendamment de l'utilisateur, en fonction des informations fournies par ce dernier.

La traduction de cet algorithme en langage Java est la suivante :

```
private String créerUneClé(Etudiant e) {
    String tmp;
    tmp = (e.quelPrénom()).charAt(0) + e.quelNom();
    return tmp.toUpperCase();
 }
```

À partir des données d'un étudiant `e` passées en paramètres, la méthode `créerUneClé()` retourne une chaîne de caractères majuscules (`tmp.toUpperCase()`), composée du premier caractère du prénom de l'étudiant (`(e.quelPrénom()).charAt(0)`), suivi de son nom (`e.quelNom()`). Remarquez que les données `nom` et `prénom` de la classe `Etudiant` sont privées et qu'il est donc nécessaire d'utiliser les méthodes d'accès en consultation (`quelPrénom()` et `quelNom()`) pour connaître le contenu de ces données. Ces méthodes, **à insérer dans le fichier** `Etudiant.java`, sont décrites comme suit :

```
public String quelNom()    {
    return nom;
}
public String quelPrénom()    {
    return prénom;
}
```

La création d'une clé peut également être réalisée simplement à partir des nom et prénom de l'étudiant, stockés, non pas dans un objet `Etudiant`, mais dans deux `String` distincts. La méthode `créerUneClé()` est alors surchargée de la façon suivante :

```
private String créerUneClé(String p, String n) {
    String tmp;
    tmp = p.charAt(0)+ n;
    return tmp.toUpperCase();
}
```

Les deux méthodes `créerUneClé()`, **à insérer dans la classe** `Classe`, sont déclarées en mode `private` car elles constituent un traitement interne propre au mode de stockage de l'information. L'application et l'utilisateur n'ont nullement besoin d'en connaître l'existence pour créer la liste des étudiants d'une classe.

Création du dictionnaire

Pour créer le dictionnaire d'une classe d'étudiants, nous devons tout d'abord définir un objet de type `Hashtable` puis stocker dans cet objet les étudiants, en les associant à leur clé. Ces deux opérations sont réalisées dans le programme suivant :

```
import java.util.*;
public class Classe  {
 private Hashtable listeClassée;
 public Classe()   {
    listeClassée = new Hashtable();
}
public void ajouteUnEtudiant() {
    Etudiant nouveau = new Etudiant();
    String clé = créerUneClé(nouveau);
```

```
      if (listeClassée.get(clé)== null) listeClassée.put(clé, nouveau);
      else System.out.println("Cet etudiant a deja ete saisi !");
  }
}
```

Ce programme est constitué des deux méthodes suivantes :

- Le constructeur `Classe()`, qui fait appel au constructeur de la classe `Hashtable` afin de déterminer l'adresse du premier élément de la `listeClassée`.
- `ajouteUnEtudiant()`, `qui` place un élément dans le dictionnaire grâce à la méthode `put(clé, nouveau)`, qui ajoute l'association `clé-nouveau` dans le dictionnaire `listeClassée`. L'objet `nouveau` est de type `Etudiant`. Il est créé par l'intermédiaire du constructeur `Etudiant()`, et `clé` est aussi un objet de type `String` calculé à partir de la méthode `créerUneClé()`.

  L'ajout successif de deux associations ayant la même `clé` a pour résultat de détruire la première association. C'est pourquoi il convient de tester, avant de placer le nouvel étudiant dans le dictionnaire, si ce dernier n'est pas déjà défini dans la `listeClassée`. C'est ce que réalise le test suivant :

```
if (listeClassée.get(clé) == null) listeClassée.put(clé, nouveau);
```

  En effet, en testant le résultat de la méthode `get(clé)`, le programme recherche dans le dictionnaire s'il existe un étudiant associé à la `clé` calculée, correspondant à l'étudiant que l'on souhaite insérer dans la liste (`nouveau`). Si cette association n'existe pas, l'élément retourné par la méthode est `null`, et l'interpréteur ajoute la nouvelle association `clé-nouveau` dans le dictionnaire. Dans le cas contraire, le programme affiche un message précisant que l'étudiant existe déjà.

Rechercher et supprimer un élément du dictionnaire

Une fois le dictionnaire réalisé, les opérations permettant la recherche ou la suppression d'un étudiant sont décrites par les méthodes suivantes, à insérer dans le fichier `Classe.java` :

```
 public void rechercheUnEtudiant(String p, String n) {
     String clé = créerUneClé(p, n);
     Etudiant eClassé = (Etudiant) listeClassée.get(clé);
     if (eClassé != null) eClassé.afficheUnEtudiant();
     else System.out.println(p + " " + n + " est inconnu ! ");
 }
 public void supprimeUnEtudiant(String p, String n) {
    String clé = créerUneClé(p, n);
    Etudiant eClassé = (Etudiant) listeClassée.get(clé);
    if (eClassé!= null)  {
        listeClassée.remove(clé);
        System.out.println(p + " " + n + " a ete supprime ");
    }
    else System.out.println(p + " " + n + " est inconnu ! ");
 }
```

Ces méthodes fonctionnent toutes deux sur le même modèle. Les nom et prénom de l'étudiant à traiter sont fournis en paramètres des méthodes afin de calculer la clé d'association. Ensuite, l'étudiant est recherché dans la liste à partir de cette clé (`get(clé)`).

S'il est trouvé, il est soit affiché (`eClassé.afficheUnEtudiant()`, pour la méthode `rechercheUnEtudiant()`), soit supprimé (`listeClassée.remove(clé)`, pour la méthode `supprimeUnEtudiant()`).

Afficher un dictionnaire

Pour afficher tous les éléments d'un dictionnaire, nous devons le parcourir élément par élément. Il existe différentes techniques pour réaliser ce parcours. Nous vous en proposons une, qui utilise un outil du langage Java, défini par la classe `Enumeration`.

Une énumération est un outil du package `java.util`, qui facilite le parcours de listes de données, quelles qu'elles soient. Ainsi, pour se déplacer dans une énumération d'objets, il suffit d'utiliser les méthodes `hasMoreElements()` et `nextElement()`. La première méthode détermine s'il existe encore des éléments dans l'énumération, tandis que la seconde permet l'accès à l'élément suivant dans l'énumération.

La méthode `afficheLesEtudiants()` utilise cette technique pour réaliser la parcours de la `listeClassée`.

```
public void afficheLesEtudiants()    {
    if(listeClassée.size() != 0)    {
        Enumeration enumEtudiant = listeClassée.keys();
        while (enumEtudiant.hasMoreElements()) {
            String clé = (String) enumEtudiant.nextElement();
            Etudiant eClassé = (Etudiant) listeClassée.get(clé);
            eClassé.afficheUnEtudiant();
        }
    }
    else System.out.println("Il n'y a pas d'etudiant dans cette liste");
 }
}
```

L'énumération est définie grâce à la méthode `keys()` de la classe `Hashtable`, qui renvoie sous forme d'énumération la liste des clés effectivement stockées. Le parcours de cette énumération est ensuite réalisé à l'aide d'une boucle `while` testant s'il existe encore des clés dans la liste (`enumEtudiant.hasMoreElements()`). Si tel est le cas, le programme passe à l'élément suivant dans la liste (`enumEtudiant.nextElement()`). Il recherche alors l'étudiant associé à cette clé (`listeClassée.get(clé)`) et l'affiche (`eClassé.afficheUnEtudiant()`).

Exemple : L'application `GestionClasse`

La gestion des étudiants d'une classe est totalement achevée en écrivant l'application `GestionClasse` comme suit :

```
public class GestionClasse  {
public static void main (String [] argument) {
    byte choix = 0 ;
    Classe C = new Classe();
```

```
    String prénom, nom;
    do  {
        System.out.println("1. Ajoute un etudiant");
        System.out.println("2. Supprime un etudiant");
        System.out.println("3. Affiche la classe");
        System.out.println("4. Affiche un etudiant");
        System.out.println("5. Sortir");
        System.out.println();
        System.out.print("Votre choix : ");
        choix  = Lire.b();
        switch (choix) {
        case 1 : C.ajouteUnEtudiant();
        break;
        case 2 : System.out.print("Entrer le prenom de l'etudiant : ");
                 prénom = Lire.S();
                 System.out.print("Entrer le nom de l'etudiant : ");
                 nom = Lire.S();
                 C.supprimeUnEtudiant(prénom, nom);
        break;
        case 3 : C.afficheLesEtudiants();
        break;
        case 4 : System.out.print("Entrer le prenom de l'etudiant : ");
                 prénom = Lire.S();
                 System.out.print("Entrer le nom de l'etudiant : ");
                 nom = Lire.S();
                 C.rechercheUnEtudiant(prénom, nom);
        break;
        case 5 : System.exit(0) ;
        default : System.out.println("Cette option n'existe pas ");
      }
    } while ( choix != 5);
 }
}
```

Chaque option du menu utilise une méthode de la classe `Classe`. Ces options offrent la possibilité d'ajouter, de supprimer et de consulter tout ou partie du dictionnaire, formé au fur et à mesure des choix de l'utilisateur. L'exécution de cette application peut avoir, par exemple, pour résultat à l'écran :

```
1. Ajoute un etudiant
2. Supprime un etudiant
3. Affiche la classe
4. Affiche un etudiant
5. Sortir
Votre choix : 1
Entrer le nom de l'etudiant : R.
Entrer le prenom de l'etudiant : Sylvain
Combien de notes pour l'etudiant  Sylvain R. :  2
Entrer la note  n° 1 :  15
Entrer la note  n° 2 :  14
1. Ajoute un etudiant
2. Supprime un etudiant
```

```
3. Affiche la classe
4. Affiche un etudiant
5. Sortir
Votre choix : 1
Entrer le nom de l'etudiant : C.
Entrer le prenom de l'etudiant : Gaelle
Combien de notes pour l'etudiant  Gaelle C. :  2
Entrer la note  n° 1 :  16
Entrer la note  n° 2 :  12
1. Ajoute un etudiant
2. Supprime un etudiant
3. Affiche la classe
4. Affiche un etudiant
5. Sortir
Votre choix : 4
Entrer le prenom de l'étudiant recherche : C.
Entrer le nom de l'étudiant recherche : Gaelle
C. Gaelle est inconnu !
1. Ajoute un etudiant
2. Supprime un etudiant
3. Affiche la classe
4. Affiche un etudiant
5. Sortir
Votre choix : 4
Entrer le prenom de l'étudiant recherche : Gaelle
Entrer le nom de l'étudiant recherche : C.
 Les notes  de Gaelle C. sont :  16.0 12.0
Sa moyenne vaut  14.0
1. Ajoute un etudiant
2. Supprime un etudiant
3. Affiche la classe
4. Affiche un etudiant
5. Sortir
Votre choix : 5
```

Lors du premier choix `4`, l'utilisateur a inversé les nom et prénom de l'étudiante. La clé qui en découle n'existe pas dans le dictionnaire. Le programme ne peut donc pas retrouver les informations concernant cette étudiante.

Ainsi, grâce aux objets de type `Hashtable`, il est possible d'organiser, sans beaucoup d'efforts de programmation, des données de façon à pouvoir rechercher, modifier ou supprimer un élément dans une liste.

Pourtant, l'application `GestionClasse` possède encore un inconvénient majeur : elle perd la mémoire… En effet, à chaque exécution, les données doivent de nouveau être saisies au clavier. Les données stockées dans la mémoire vive de l'ordinateur se perdent à l'arrêt du programme. Pour corriger ce défaut, le programme doit pouvoir enregistrer les informations traitées dans un fichier sur le disque dur. Cet enregistrement des données est aussi appelé archivage de données.

L'archivage de données

La notion d'archivage de données est très importante puisque, grâce à elle, les informations traitées sont stockées sous forme de fichiers sur le disque dur. Les informations ainsi stockées perdent leur volatilité et peuvent être réutilisées après un arrêt momentané du programme ou de l'ordinateur. Pour archiver des données, le langage Java utilise le concept de flux, ou, en anglais, `stream`.

La notion de flux

Lorsque nous communiquons des informations à l'ordinateur, nous effectuons une opération d'entrée (*voir le Chapitre 2, « Communiquer une information »*). Cette communication utilise un flux entrant, qui est, en quelque sorte, la concrétisation informatique du courant électrique passant du clavier à la mémoire vive de l'ordinateur. Symétriquement, il existe un flux sortant, permettant de faire passer une information stockée en mémoire vive à l'écran de l'ordinateur.

Dans le jargon informatique, on dit que les flux reliant la mémoire vive à l'écran ou au clavier utilisent des flux standards d'entrée-sortie. De façon similaire, il existe d'autres types de flux, qui relient la mémoire vive, non plus à l'écran ou au clavier, mais au disque dur de l'ordinateur. Ce sont les flux de fichiers.

Ces flux sont aussi caractérisés par leur direction, entrante ou sortante. Un flux de fichier sortant est un flux d'écriture qui réalise la création et l'enregistrement de données dans un fichier. Symétriquement, un flux de fichier entrant est un flux de lecture qui permet l'initialisation des variables ou objets du programme en mémoire vive, grâce aux valeurs précédemment enregistrées sur le disque dur.

Ces flux sont définis à travers des objets prédéfinis du langage Java (package `java.io`). Il en existe un très grand nombre, offrant tous des outils permettant le stockage et le traitement de données sous diverses formes.

Notre objectif n'est pas de les décrire tous, même succinctement, mais de présenter concrètement deux techniques d'archivage afin d'en comprendre les différents mécanismes. C'est pourquoi notre étude porte sur les fichiers stockant l'information sous la forme de caractères (*voir, ci-dessous, la section « Les fichiers textes »*), ainsi que sur les fichiers stockant des objets (*voir la section « Les fichiers d'objets »*)

Les fichiers textes

Puisqu'il existe deux façons d'accéder à un fichier (lecture ou écriture), les outils proposés par le langage Java reproduisent ces deux modes d'accès. Pour manipuler des fichiers, nous devons donc déclarer deux objets différents, qui vont nous permettre de lire ou d'écrire dans un fichier.

Ainsi, la déclaration :

```
BufferedWriter fW;
```

définit un objet `fW` de type `BufferedWriter`, utilisé pour écrire (`Writer`), c'est-à-dire enregistrer des données dans un fichier.

Par contre, la déclaration :

```
BufferedReader fR;
```

définit un objet fR de type BufferedReader, utilisé pour lire (Reader) les données contenues dans un fichier afin de les placer dans des variables (en mémoire vive). Ces objets et les méthodes associées sont définis dans le package java.io. Il convient donc de placer l'instruction import java.io.* ; en en-tête des classes qui font appel à ces outils.

Exemple : Une classe pour lire et écrire du texte

L'objectif de cet exemple est de créer une classe Fichier composée d'outils simplifiant la manipulation des fichiers en lecture et en écriture. Les données de cette classe définissent deux objets de type BufferedWriter et BufferedReader et d'une variable de type char, qui mémorise le mode de traitement utilisé (lecture ou écriture).

```
import java.io.*;
public class Fichier {
  private BufferedWriter fW;
  private BufferedReader fR;
  private char mode;
}
```

D'une façon générale, les traitements sur fichiers se déroulent en trois temps : ouverture du flux, puis traitement des données parcourant le flux et, pour finir, fermeture du flux. Chacune de ces étapes est décrite au cours des sections suivantes.

Ouverture du flux

Pour écrire ou lire dans un fichier, il nécessaire, avant tout, d'ouvrir le flux en indiquant si ce flux est entrant (lecture) ou sortant (écriture). La méthode ouvrir() décrite ci-dessous réalise cette opération. Elle doit être insérée dans la classe Fichier.

```
public void ouvrir(String nomDuFichier, String s) throws IOException    {
    mode = (s.toUpperCase()).charAt(0);
    if (mode == 'R' || mode == 'L')
       fR  = new BufferedReader(new FileReader(new File(nomDuFichier)));
    else    if (mode == 'W' || mode == 'E')
       fW = new BufferedWriter(new FileWriter(new File(nomDuFichier)));
}
```

Les deux points importants suivants sont à observer dans l'en-tête de la méthode ouvrir() :

- Le premier concerne le terme throws IOException, dont la présence est obligatoire pour toutes les méthodes qui manipulent des opérations d'entrée-sortie. Succinctement, cette clause indique au compilateur que la méthode ainsi définie est susceptible de traiter ou de propager une éventuelle erreur, du type IOException.., qui pourrait apparaître en cours d'exécution.

✔ Vour plus de précisions, voir la section *« Gérer les exceptions »*, en fin de chapitre.

- Le second point important est relatif aux informations transmises à la méthode `ouvrir()`. Le premier paramètre spécifie le nom du fichier auquel est associé le flux, tandis que le second indique le mode d'ouverture du flux (entrant ou sortant). Ce paramètre peut prendre différentes valeurs, telles que `"Ecriture"`, `"E"`, `"Write"` ou encore `"W"`, pour le mode sortant, et `"Lecture"`, `"L"`, `"Read"` ou encore `"R"`, pour le mode entrant. Ces mots peuvent être écrits indifféremment en majuscules ou en minuscules. En effet, la variable d'instance `mode` est initialisée à partir du premier caractère (`charAt(0)`) du paramètre `s` et est automatiquement transformée en majuscules (`s.toUpperCase()`).

Cela fait, le flux est ouvert en lecture ou en écriture en fonction de la variable d'instance `mode`. Ainsi :

- Si `mode` vaut `L` ou `R`, l'ouverture du fichier est réalisée en lecture grâce à l'instruction :

```
fR = new BufferedReader(new FileReader(new File(nomDuFichier)));
```

  Cette instruction relativement déconcertante pour le programmeur débutant réalise plusieurs opérations afin de déterminer où se situe le début du fichier spécifié en paramètre.

  La première opération `new File(nomDuFichier)` permet d'obtenir une représentation logique du fichier (existe-t-il ou non sur le disque dur ?). Ensuite, l'appel au constructeur `FileReader()` permet l'ouverture du fichier en lecture caractère par caractère. Il fournit en retour l'adresse du début du fichier.

  Cependant, ce mode de lecture n'autorise pas la lecture de plusieurs caractères à la fois. C'est pourquoi il est nécessaire de faire appel à un troisième constructeur, `BufferedReader()`, qui permet la lecture du fichier ligne par ligne. L'adresse du début du fichier est alors mémorisée, grâce au signe d'affectation, dans l'objet `fR`.

- Si `mode` vaut `E` ou `W`, l'ouverture du fichier est réalisée en écriture grâce à l'instruction :

```
fW = new BufferedWriter(new FileWriter(new File(nomDuFichier)));
```

Les opérations réalisées sont équivalentes à celles décrites ci-dessus, en remplaçant le mode lecture par le mode écriture. Cependant,

- Si le fichier spécifié en paramètre n'existe pas, et :
  - Si le chemin d'accès à ce fichier dans l'arborescence du disque est valide, alors le fichier est créé, et l'adresse du début du fichier est stockée dans l'objet `fW`.
  - Si le chemin d'accès n'est pas valide, le fichier n'est pas créé, et une erreur du type `FileNotFoundException` est détectée.
- Si le fichier existe, il est ouvert, et son contenu est totalement effacé. L'adresse du début du fichier est alors stockée dans l'objet `fW`.

Traitement du fichier

Une fois le fichier ouvert, les traitements réalisables sur lui sont l'écriture et la lecture de données dans le fichier.

- L'**écriture** dans un fichier est réalisée par la méthode suivante :

```
public void ecrire(int tmp) throws IOException {
    String chaine = "";
    chaine = chaine.valueOf(tmp);
    if (chaine != null)   {
        fW.write(chaine,0,chaine.length());
        fW.newLine();
    }
}
```

  La méthode `ecrire()` prend en paramètre la valeur à enregistrer dans le fichier. Comme il s'agit d'un entier et que le fichier est un fichier texte, la valeur stockée dans `tmp` est convertie en `String` grâce à l'instruction `chaine = chaine.valueOf(tmp)` .

  Ensuite, l'écriture de cette chaîne dans le fichier est réalisée par l'instruction `fW.write(chaine, 0, chaine.length())`. La méthode `write()` envoie dans le flux `fW` la chaîne spécifiée en premier paramètre. Les deuxième et troisième paramètres précisent respectivement à partir de quel caractère (`0`) commence l'écriture dans le fichier et combien de caractères `(chaine.length)` sont écrits. Pour notre exemple, l'intégralité de la `chaine` est écrite dans le fichier.

  Pour finir, la méthode `newLine()`envoie dans le flux `fW` un caractère permettant de passer à la ligne suivante du fichier.

- La **lecture** dans un fichier est décrite par la méthode :

```
public String lire() throws IOException {
    String chaine = fR.readLine();
    return chaine;
}
```

  L'opération de lecture est réalisée par la méthode `readLine()`, qui envoie tout d'abord la ligne lue sur le flux `fR` puis passe automatiquement à la ligne suivante dans le fichier. La chaîne de caractères `chaine` récupère alors la suite des caractères lus. Pour finir, la chaîne est retournée à la méthode appelante.

Fermeture du flux

Une fois que tous les traitements ont été réalisés, le flux peut être naturellement fermé grâce à la méthode :

```
public void fermer() throws IOException {
    if (mode == 'R' || mode == 'L') fR.close();
    else    if (mode == 'W' || mode == 'E') fW.close();
}
```

Suivant le mode d'ouverture spécifié par la variable d'instance `mode` (initialisée lors de l'exécution de la méthode `ouvrir()`), le flux `fR` ou `fW` est fermé grâce à la méthode `close()`.

Exemple : L'application `GestionFichier`

L'application suivante utilise les méthodes décrites ci-dessus pour créer et manipuler un fichier dont le nom est saisi au clavier :

```
public class GestionFichier {
 public static void main (String [] arg) throws IOException  {
    Fichier f = new Fichier();
    System.out.print(" Entrer le nom du fichier : ");
    String nomFichier = Lire.S();
    f.ouvrir(nomFichier, "Ecriture");
    for (int i = 0; i < 5; i++)  f.ecrire(i);
    f.fermer();

    f.ouvrir(nomFichier,"Lecture");
    String chaine ="";
    do {
        chaine = f.lire();
        if (chaine != null) System.out.println(chaine);
    } while (chaine != null);
    f.fermer();
 }
}
```

L'instruction `f.ouvrir(nomFichier, "Ecriture")` ouvre le fichier `nomFichier` en écriture afin d'y écrire une suite de valeurs entières (`f.ecrire(i)`) comprises entre `0` et `4`. Le fichier est fermé (`f.fermer()`) après exécution de la boucle `for`.

Pour vérifier que les opérations d'écriture se sont bien déroulées, le fichier est ouvert en lecture (`f.ouvrir(nomFichier,"Lecture")`) et, grâce à une boucle `do…while`, chaque ligne du fichier est lue par `f.lire()` et mémorisée dans une variable `chaine` afin d'être affichée. La lecture de ce fichier prend fin lorsqu'une chaîne `null` est détectée (`while (chaine != null)`). Le fichier peut alors être fermé (`f.fermer()`).

L'exécution de cette application a pour résultat à l'écran :

```
Entrer le nom du fichier : Valeurs.txt
0
1
2
3
4
```

Le fichier `Valeurs.txt` est créé dans le même répertoire que celui où se trouve l'application `GestionFichier.class`. Comme il s'agit d'un fichier texte, il peut être ouvert par n'importe quel éditeur de texte (`WordPad` sous Windows, `vi` sous Unix ou encore `TeachText` sous Mac OS). C'est là un des intérêts des fichiers textes.

Remarquez, cependant, que les données manipulées par un programme ne se résument généralement pas à de simples valeurs entières. Le plus souvent, une application travaille avec des objets complexes, mêlant plusieurs types de données. C'est pourquoi il est intéressant de pouvoir archiver, non pas la suite des données relatives à un objet, ligne par ligne, mais l'objet lui-même en tant que tel. Cette technique est examinée à la section suivante.

Les fichiers d'objets

Le langage Java propose des outils permettant le stockage ainsi que la lecture d'objets dans un fichier. Ces outils font appel à des mécanismes appelés mécanismes de sérialisation. Ils utilisent des flux spécifiques, définis par les classes `ObjectOutputStream` et `ObjectInputStream`, du package `java.io`.

La sérialisation des objets

Un objet est sérialisé afin de pouvoir être transporté sur un flux de fichier, entrant ou sortant. Grâce à cette technique, un objet peut être directement stocké dans un fichier (écriture) ou reconstruit à l'identique en mémoire vive par lecture du fichier.

Les mécanismes de sérialisation-désérialisation sont fournis par l'intermédiaire des classes `ObjectOutputStream` et `ObjectInputStream`, grâce aux méthodes `writeObject()` (sérialisation) et `readObject()` (désérialisation). Ces outils sont applicables à tous les objets prédéfinis du langage Java, tels que les `String`, les vecteurs (`Vector`) ou encore les dictionnaires (`Hashtable`).

Lorsque vous souhaitez sérialiser un objet dont le type est défini par le programmeur, il est nécessaire de rendre cet objet sérialisable. Pour cela, il suffit d'indiquer au compilateur que la classe autorise la sérialisation, en utilisant la syntaxe suivante :

```
public class Exemple implements Serializable {
// Données et méthodes
}
```

De cette façon, tous les objets déclarés de type `Exemple` peuvent être lus ou écrits dans des fichiers d'objets. Remarquez que l'objectif de la sérialisation est de placer, dans un flux, toutes les informations relatives à un objet. Par conséquent, seules les variables d'instance sont prises en compte lors d'une sérialisation, alors que les variables de classes (définies en `static`) ne peuvent être sérialisées. En effet, une variable de classe est commune à tous les objets de l'application et non pas spécifique d'un seul objet.

Exemple : Archiver une classe d'étudiants

Pour bien comprendre comment utiliser ces outils d'archivage d'objets, modifions l'application `GestionClasse` de sorte qu'elle puisse lire et stocker automatiquement l'ensemble des données du dictionnaire `listeClassée` dans un fichier, portant le nom de `Classe.dat`.

Nous devons tout d'abord rendre sérialisable les objets que nous souhaitons sauvegarder. C'est pourquoi il convient de modifier les en-têtes des classes `Etudiant` et `Classe` de la façon suivante :

```
public class Etudiant implements Serializable {
// voir la section "Les dictionnaires"
}
public class Classe implements Serializable {
// voir la section "Les dictionnaires"
}
```

À défaut, vous obtenez une erreur d'exécution du type NotSerializableException, indiquant que l'objet de type Etudiant ou Classe ne peut être sérialisé.

Ensuite, les opérations d'archivage d'objets utilisent les mêmes concepts que ceux décrits à la section précédente, à savoir ouverture du fichier, puis lecture ou écriture des objets et, pour finir, fermeture du fichier. C'est pourquoi nous allons modifier la classe Fichier pour y manipuler, non plus des fichiers textes, mais des fichiers d'objets.

```
import java.io.*;
public class FichierEtudiant {
 private ObjectOutputStream ofW;
 private ObjectInputStream ofR;
 private String nomDuFichier = "Classe.dat";
 private char mode;
}
```

Les données de la classe FichierEtudiant sont deux objets représentant des flux d'écriture (ofW), de lecture (ofR) d'objets, ainsi qu'un caractère (mode) représentant le type d'ouverture du fichier et une chaîne de caractères (nomDuFichier), où se trouve mémorisé le nom de fichier de sauvegarde des données (Classe.dat).

Ouverture du flux (entrant ou sortant)

```
public void ouvrir(String s) throws IOException    {
   mode = (s.toUpperCase()).charAt(0);
   if (mode == 'R' || mode == 'L')
     ofR  = new ObjectInputStream(new FileInputStream(nomDuFichier));
   else    if (mode == 'W' || mode == 'E')
     ofW  = new ObjectOutputStream(new FileOutputStream(nomDuFichier));
}
```

L'ouverture du fichier Classe.dat en lecture est réalisée grâce aux constructeurs des classes FileInputStream et ObjectInputStream, alors que l'ouverture en écriture est effectuée par les constructeurs ObjectOutputStream() et FileOutputStream(). En résultat, les flux ofW et ofR contiennent les adresses de début de fichier.

Traitement du fichier

L'objectif est d'archiver l'ensemble des données relatives à une classe d'étudiants. La méthode ecrire() prend en paramètre un objet tmp de type Classe, de sorte que l'information lui soit transmise depuis l'application GestionClasse. L'objet transmis est alors archivé grâce à la méthode writeObject(tmp).

```
public void ecrire(Classe tmp) throws IOException {
   if (tmp != null)  ofW.writeObject(tmp);
}
```

Inversement, la méthode lire() lit l'objet stocké dans le fichier Classe.dat et le transmet en retour à l'application GestionClasse sous forme d'objet de type Classe. L'en-tête de la méthode a pour type le type Classe. L'objet retourné est lu grâce à la méthode readObject().

```
public Classe lire() throws IOException, ClassNotFoundException  {
   Classe tmp = (Classe) ofR.readObject();
   return tmp;
}
```

Observons que :

- La méthode `lire()` traite obligatoirement un nouveau type d'exception : `ClassNotFoundException`. En effet, la méthode `readObject()` transmet ce type d'exception lorsque le fichier lu ne contient pas d'objet mais tout autre chose.

> ✔ Pour plus de précisions sur la gestion des exceptions, voir la section *« Gérer les exceptions »*, en fin de chapitre.

- La méthode `readObject()` lit sur le flux un objet, quel que soit son type. Il est donc nécessaire de spécifier, par l'intermédiaire d'un « cast », le format de l'objet lu (*voir, au Chapitre 1, « Stocker une information », la section « La transformation de type »*). Pour notre exemple, l'objet lu est transmis à l'objet `tmp` par l'intermédiaire d'un cast `(Classe)`, qui réalise la transformation de l'objet au bon format.

Fermeture du flux

La fermeture d'un flux est réalisée par la méthode `close()`, de la même façon qu'un flux de fichier texte.

```
public void fermer() throws IOException {
    if (mode == 'R' || mode == 'L') fRo.close();
    else    if (mode == 'W' || mode == 'E') fWo.close();
}
```

Exemple : L'application GestionClasse

L'application `GestionClasse` a pour contrainte de réaliser les actions suivantes :

- Une lecture automatique du fichier « `classe.dat` » dès l'ouverture du programme afin d'initialiser l'objet `C` (type `Classe`) à la liste d'étudiants saisie lors d'une précédente exécution.
- Une sauvegarde automatique dans le fichier « `classe.dat` » lorsque l'utilisateur choisit de sortir du programme.

Ces deux contraintes sont réalisées par l'application suivante :

```
import java.io.*;
public class GestionClasse  {
public static void main (String [] argument)
           throws IOException, ClassNotFoundException {
    byte choix = 0 ;
    Classe C = new Classe();
    FichierEtudiant F = new FichierEtudiant();
    F.ouvrir("Lecture");
    C = F.lire();
    F.fermer();
    String prénom, nom;
    do {
```

```
            System.out.println("1. Ajoute un etudiant");
            System.out.println("2. Supprime un etudiant");
            System.out.println("3. Affiche la classe");
            System.out.println("4. Affiche un etudiant");
            System.out.println("5. Sortir");
            System.out.println();
            System.out.print("Votre choix : ");
            choix  = Lire.b();
            switch (choix) {
                // pour les options 1, 2, 3, 4 voir
                // Exemple : Créer un dictionnaire d'étudiants
                case 1 : // Ajoute un etudiant
                case 2 : // Supprime un etudiant
                case 3 : // Affiche les etudiants
                case 4 : // Affiche un etudiant
                case 5 :
                    System.out.println("Sauvegarde des donnees dans
                    ➥Classe.dat");
                    F.ouvrir("Ecriture");
                    F.ecrire(C);
                    F.fermer();
                    System.exit(0) ;
                break;
                default :  System.out.println("Cette option n'existe pas ");
            }
        } while (choix != 5);
    }
} // Fin de la classe GestionClasse
```

L'exécution de cette application montre qu'une difficulté subsiste. En effet, lors de la toute première exécution du programme, l'interpréteur affiche le message suivant :

```
java.io.FileNotFoundException:
Classe.dat (Le fichier spécifié est introuvable)
    at java.io.FileInputStream.open(Native Method)
    at java.io.FileInputStream.<init>(FileInputStream.java:68)
    at FichierEtudiant.ouvrir(FichierEtudiant.java:14)
    at GestionClasse.main(Compiled Code)
```

L'erreur `FileNotFoundException` est transmise à la méthode `main()` *via* la méthode `FileInputStream.open()` grâce à la clause `throws IOException`.

En effet, le fichier `Classe.dat` n'existe pas encore, puisque c'est la première fois que le programme est exécuté. L'option 5 n'a pu être exécutée, et aucune sauvegarde n'a donc été réalisée. Tant que le programme ne peut être exécuté dans son intégralité, aucun fichier de sauvegarde ne peut être créé.

Pour contourner cet obstacle, la solution consiste à empêcher les erreurs de remonter d'une méthode à l'autre grâce à la clause `throws`, tout en gérant de façon explicite chaque erreur qui pourrait survenir. Cette solution est examinée à la section suivante.

Gérer les exceptions

Plutôt que de laisser l'erreur se propager (avec la clause `throws`), le langage Java propose des outils de capture des erreurs afin de les traiter directement à l'intérieur des méthodes susceptibles de les détecter. Cette capture est réalisée par l'intermédiaire des instructions `try…catch`.

La méthode `ouvrir()`

Examinons le mécanisme de ces instructions sur la méthode `ouvrir()`, proposée à la section précédente. Comme observé précédemment, cette méthode pose problème, puisqu'elle propage l'erreur `FileNotFoundException` lors de la toute première exécution de l'application `GestionClasse`. Pour éviter cette propagation, l'idée est de placer le couple d'instructions `try…catch` de la façon suivante :

```
public boolean ouvrir(String s) {
   try {
       mode = (s.toUpperCase()).charAt(0);
       if (mode == 'R' || mode == 'L')
         fRo = new ObjectInputStream(new FileInputStream(nomDuFichier));
       else     if (mode == 'W' || mode == 'E')
       fWo = new ObjectOutputStream(new FileOutputStream(nomDuFichier));
       return true;
   }
   catch (IOException e)    {
       return false;
   }
}
```

La méthode s'exécute alors de la façon suivante comme expliqué ci-après.

Les instructions qui composent le bloc `try` (en français `essayer`) sont exécutées.

- Si aucune erreur n'est transmise par les différents constructeurs qui réalisent l'ouverture du fichier, le programme sort de la méthode `ouvrir()` en retournant un booléen de valeur égale à `true`.
- Si une erreur est propagée par l'un des constructeurs, les instructions placées dans le bloc `catch` (capture) sont exécutées, à condition que l'erreur détectée soit du même type que celui placé entre parenthèses derrière le terme `catch`. L'erreur `FileNotFoundException` étant du type `IOException`, le programme sort de la méthode `ouvrir()`, en retournant un booléen de valeur égale à `false`. Aucun fichier n'est donc ouvert.

Puisque la méthode `ouvrir()` capture et traite elle-même les erreurs éventuelles, la présence de la clause `throws` devient inutile, et elle n'apparaît plus dans l'en-tête de la méthode.

Grâce au traitement des erreurs en interne, les instructions relatives à l'ouverture du fichier en lecture dans l'application `GestionClasse` peuvent être modifiées de la façon suivante :

```
FichierEtudiant F = new FichierEtudiant();
if (F.ouvrir("L")) {
    C = F.lire();
    F.fermer();
}
```

Le fichier est ouvert grâce à l'instruction `F.ouvrir()`. Si le résultat de la méthode vaut `true`, cela signifie que le fichier `classe.dat` existe et est ouvert. Les instructions de lecture du fichier situées dans le bloc `if` peuvent être exécutées. À l'inverse, si le résultat vaut `false`, aucune instruction n'est exécutée. Le menu permettant la saisie de nouveaux étudiants peut alors être affiché.

La méthode lire()

La méthode `lire()` est susceptible de lever plusieurs types d'exception, *via* la méthode `readObject()`. En effet, cette dernière est susceptible de détecter des erreurs du type `IOException` ou `ClassNotFoundException`.

La capture de ces exceptions est réalisée en définissant autant de blocs `catch` qu'il y a d'erreurs détectées. La méthode `lire()` traite ces erreurs de la façon suivante :

```
public Classe lire() {
    try {
        Classe tmp = (Classe) fRo.readObject();
        return tmp;
    }
    catch (IOException e) {
        System.out.println(nomDuFichier + " : Erreur de lecture ");
    }
    catch (ClassNotFoundException e) {
        System.out.println(nomDuFichier + " n'est pas du bon format ");
    }
    return null;
}
```

Il est ainsi possible de définir deux blocs `catch` successifs, paramétrés en fonction des types d'erreurs susceptibles d'être détectés. Le programme réagit différemment suivant l'erreur capturée.

Résumé

La programmation **dynamique** permet la gestion d'un nombre indéterminé d'objets, en réservant des espaces mémoire au fur et à mesure des besoins de l'utilisateur.

Pour ce faire, le langage Java propose différents outils, tels que les objets de type `Vector` ou encore de type `Hashtable`.

Les objets de type `Vector` autorisent la création d'une liste, par ajout de données au fur et à mesure des besoins de l'utilisateur. Les données sont, en général, enregistrées dans leur ordre d'arrivée. Un indice géré par l'interpréteur permet de retrouver l'information.

Pour utiliser un vecteur, il est nécessaire de le déclarer de la façon suivante :

```
Vector liste = new Vector() ;
```

Pour **ajouter un objet à la liste**, il suffit d'écrire `liste.addElement(objet)`. Il n'est pas possible d'ajouter une valeur autre qu'un objet (telle que les variables de type `int`, par exemple).

Lorsque que l'objet est inséré dans la liste, la taille de cette dernière est augmentée de un. La méthode `size()` calcule le nombre d'éléments dans la liste, et la méthode `elementAt(indice)` permet de retrouver l'objet stocké à l'`indice` spécifié en paramètre.

La classe `Vector` étant définie dans le package `java.util`, il convient de placer l'instruction `import.java.util.*;` en tête du fichier. En effet, si cette instruction fait défaut, le compilateur détecte une erreur du type `Class Vector not found`.

La recherche d'éléments complexes dans une liste est plus rapide lorsque les données sont organisées, non plus par rapport à un indice, mais par rapport à une clé explicite. Les objets de type `Hashtable` proposent ce type d'organisation des données. Pour cela, il suffit de déclarer une liste comme :

```
Hashtable liste = new Hashtable () ;
```

Les méthodes `put(clé, objet)` et `get(clé)` permettent respectivement de placer dans la liste (**dictionnaire**) l'association `clé-objet` et de retrouver l'objet associé à la `clé` spécifiée en paramètre.

Pour éviter que les données stockées en mémoire vive de l'ordinateur ne se perdent à l'arrêt de l'application, il est nécessaire de les archiver sous forme de fichiers sur le disque dur. Pour cela, le langage Java utilise le concept de flux de fichier (en anglais `stream`), qui est, en quelque sorte, la concrétisation informatique du courant électrique passant de la mémoire vive au disque dur de l'ordinateur.

Il existe différents types de **flux de fichiers** :

- D'une part, les flux **entrant**, pour lire les données sur le disque dur et les placer en mémoire vive, et les flux **sortant**, qui écrivent les données de la mémoire vive sur le disque dur.
- D'autre part, les fichiers de type texte (`BufferedWriter`, `BufferedReader`), qui ne font que manipuler des données de type `String`, et les fichiers d'objets (`ObjectOutputStream`, `ObjectInputStream`), qui manipulent tout type d'objet.

D'une façon générale, les traitements sur fichiers se déroulent en trois temps : **ouverture** du flux, **traitement** des données parcourant le flux, puis **fermeture** du flux. Lorsqu'un fichier est ouvert en écriture :

- Si le fichier n'existe pas, et :
  - Si le chemin d'accès à ce fichier dans l'arborescence du disque est valide, alors le fichier est créé.
  - Si le chemin d'accès n'est pas valide, alors le fichier n'est pas créé et une erreur du type `FileNotFoundException` est détectée.
- Si le fichier existe, il est ouvert, et son contenu est **totalement effacé**.

Lorsqu'une erreur est détectée par les méthodes associées au flux, le couple d'instructions `try…catch` permet la capture de l'exception afin de lui associer un traitement spécifique.

Exercices

Comprendre les vecteurs

10.1 L'objectif est de stocker les notes d'un étudiant sous la forme d'un vecteur.

a. Définissez un objet note de type `Vector` comme variable d'instance de la classe `Etudiant`.

b. Modifiez le constructeur de la classe `Etudiant` afin de saisir les notes et de les placer dans le vecteur.

Prenez garde que seul un objet peut être stocké dans un vecteur. Une note, étant de type `double` (type simple), ne peut pas être directement placée dans le vecteur. Il est nécessaire de la transformer en objet de type `Double`. L'appel au constructeur de la classe `Double` permet cette transformation.

Par exemple, l'instruction `new  Double(Lire.d())` permet la transformation directe d'une valeur `double` saisie au clavier en un objet de type `Double`.

c. La méthode `calculMoyenne()` doit calculer la moyenne des notes à partir des notes saisies dans le constructeur. Le programme doit, par conséquent, parcourir l'ensemble du vecteur `note` afin d'en calculer la somme. Puisque ces valeurs sont stockées sous la forme d'objets `Double`, il est nécessaire, pour réaliser ce calcul, de les transformer de nouveau en type simple `double`. Pour cela, il convient d'appliquer la méthode `doubleValue()` à l'objet de type `Double`.

d. Dans la méthode `afficheUnEtudiant()`, modifiez l'affichage des notes en parcourant, non plus le tableau, mais le vecteur `note`.

Comprendre les dictionnaires

10.2 L'objectif est d'écrire une méthode `modifieUnEtudiant()` qui modifie les notes d'un étudiant stocké dans un dictionnaire. Cette méthode fonctionne dans l'ensemble comme la méthode `ajouteUnEtudiant()` (*voir la section « Les dictionnaires » de ce chapitre*).

a. Cependant, la méthode doit connaître les `nom` et `prénom` de l'étudiant à modifier. Ces données lui sont transmises par paramètre.

b. Ensuite, connaissant les `nom` et `prénom`, le programme calcule la `clé` et vérifie si l'étudiant existe dans la liste.

c. S'il existe, la modification consiste à lui donner de nouvelles `notes`. Pour cela, l'idée est d'écrire un deuxième constructeur `Etudiant()`, dont les paramètres sont les `nom` et `prénom` de l'étudiant. Le corps du constructeur ne fait ensuite que stocker dans les variables d'instance appropriées les `nom` et `prénom` passés en paramètres, sans avoir à les ressaisir, puis saisir les nouvelles `notes` et enfin calculer la `moyenne`.

d. Modifiez l'application `GestionClasse` de façon à intégrer au menu cette nouvelle option.

Gérer les erreurs

10.3 L'objectif est de capturer toutes les erreurs (`IOException`) possibles dans la classe `FichierEtudiant` décrite au cours de ce chapitre.

a. Reprenez la classe `FichierEtudiant`, et gérez la détection des erreurs pour les méthode `fermer()` et `ecrire()`, en définissant des blocs `catch` et `try` appropriés.

b. Lorsque toutes les méthodes de la classe `FichierEtudiant` gèrent les exceptions, plus aucune clause `throws` ne doit apparaître sur l'en-tête des méthodes, y compris pour la méthode `main()` de l'application `GestionClasse`. Modifiez l'application `GestionClasse` en tenant compte de cette remarque.

Le projet « Gestion d'un compte bancaire »

Les comptes sous forme de dictionnaire

La classe `ListeCompte`

En reprenant la classe `Classe`, présentée au cours de ce chapitre, écrire la classe `ListeCompte` dont la donnée est une liste de type `Hashtable`. La classe `ListeCompte` est composée des méthodes suivantes :

a. `ListeCompte()`, qui fait appel au constructeur de la classe `Hashtable`.

b. `ajouteUnCompte(String t)`, qui permet la création d'un compte courant, joint ou d'épargne. Afin de faire appel au constructeur approprié (`Compte()` ou `CpteEpargne()`), faire passer en paramètre de la méthode `ajouteUnCompte()` une chaîne de caractères spécifiant le type du compte à créer. Par exemple, lorsque le paramètre de la méthode vaut `"E"`, un compte d'épargne est créé, alors que s'il vaut `"A"` (comme Autre), un compte ordinaire est créé.

Lorsque le compte est créé, insérez-le dans le dictionnaire, en prenant comme clé d'association son numéro de compte.

c. `ajouteUneLigne()`, qui ajoute une ligne au compte dont le numéro est spécifié en paramètre de la méthode. Pour cela, faites appel à la méthode `créerLigne()` de la classe `Compte`.

d. Les méthodes `rechercheUnCompte()`, `supprimeUnCompte()` et `afficheLesComptes()` sont à écrire en s'inspirant des méthodes équivalentes de la classe `Classe`.

L'application `Projet`

Dans l'application `Projet`, déclarer l'objet `C` comme étant du type `ListeCompte`. Puis,

a. Dans chaque option du menu, faire appel aux méthodes de la classe `ListeCompte`.

b. Lors de l'ajout d'un compte, ne pas omettre de spécifier en paramètre le type du compte (`"A"`, ou `"E"`).

c. Ajouter l'option de suppression d'un compte (option `5`) et l'affichage de la liste de tous les comptes (option `3`). Modifier l'affichage du menu et le `switch` de façon à tenir compte de ces nouvelles options. Remarquez qu'il n'est plus besoin de tester

l'existence du compte avant de l'afficher ou de le supprimer, puisque ce sont les méthodes de la classe `ListeCompte` qui s'en chargent directement.

La sauvegarde des comptes bancaires

La classe `FichierCompte`

Pour sauvegarder les données saisies pour chaque compte, reprendre la classe `FichierEtudiant` décrite dans ce chapitre :

a. Modifier le nom de la classe par `FichierCompte`, et remplacer le nom du fichier de sauvegarde par « `Compte.dat` ».

b. Dans les méthodes `lire()` et `ecrire()`, remplacer l'objet lu ou écrit par un objet de type `ListeCompte`.

c. Ne pas oublier de rendre sérialisable l'ensemble des classes nécessaires à la construction de la liste des comptes.

L'application `Projet`

Modifier l'application, de façon à :

a. lire le fichier « `Compte.dat` » avant de proposer l'ajout, la suppression ou l'affichage des comptes ;

b. réaliser une sauvegarde automatique à la sortie du programme (option `6`).

La mise en place des dates dans les lignes comptables

Chaque ligne comptable est définie par un ensemble de données, dont la date de réalisation de l'opération. Pour l'instant, cette date est saisie sous forme d'un `String`, sans aucun contrôle sur le format réellement saisi (jour/mois/an). L'objectif est d'écrire une méthode qui vérifie si les valeurs saisies correspondent au format demandé.

Rechercher des méthodes dans les différents packages

Pour effectuer ce contrôle, le langage Java propose un certain nombre d'outils définis dans les packages du JDK. En particulier, il existe des outils qui transforment une chaîne de caractères en objet `Date`. Cette transformation est réalisée à partir d'un format défini par le programmeur.

Pour trouver ces différents outils, les deux solutions suivantes sont possibles :

a. Soit rechercher dans l'arborescence du JDK fourni avec le CD-Rom (`jdk1.3\docs\api\java`) tous les fichiers contenant le mot `Date` afin de déterminer les différents packages concernés par ce type d'information. Puis, pour tous les fichiers trouvés, examiner les différentes méthodes proposées, de façon à trouver celle susceptible de répondre à votre attente.

b. Soit se connecter sur Internet, par exemple à l'adresse http://forum2.java.sun.com/forum, de façon à y rechercher des exemples utilisant des objets de type `Date`.

Écrire la méthode `contrôleDate()`

L'algorithme permettant le contrôle du format de la date est le suivant :

a. Saisir une date comme une suite de caractères (`String`).

b. Traduire cette chaîne en objet `Date` grâce aux méthodes trouvées à l'étape précédente.

c. Capturer les erreurs propagées par cette méthode afin d'incrémenter un compteur d'essai de saisie de la date.

d. Répéter ces deux derniers points tant que la date n'est pas correctement traduite (l'objet `date` restant égal à `null`). Au bout de trois essais, la date est initialisée à la date courante du système de l'ordinateur.

e. En sortie de boucle, la date correspond au format demandé. Elle peut être traduite en type `String`, pour être ensuite stockée dans la donnée `date`, de la classe `LigneComptable`.

11

Dessiner des objets

Le langage Java s'est surtout fait connaître en proposant pour Internet des outils de développement d'applications graphiques multiplates-formes, c'est-à-dire fonctionnant sur des ordinateurs de tout type. Ces programmes sont exécutés à travers un navigateur Web de façon transparente pour l'internaute, que l'ordinateur utilisé soit un Mac, un PC ou une station Unix.

Ces applications utilisent des composants graphiques définis dans la librairie graphique AWT *(Abstract Windowing Toolkit)*. Dans ce chapitre, nous étudions d'abord, à la section « La librairie AWT », comment utiliser les outils de ce package. Nous abordons ensuite, à la section « Les événements », la gestion des événements en analysant comment associer une action, ou un comportement, à un composant graphique. Pour finir, nous construisons, à la section « Les applets », une application directement exécutable par un navigateur Web.

La librairie AWT

La librairie AWT est un package du JDK *(Java Development Kit)* qui propose un ensemble d'outils de création d'applications graphiques, c'est-à-dire d'applications dont le mode de communication avec l'utilisateur s'établit à travers des éléments graphiques, tels que boutons, menus, fenêtres, etc.

Notre objectif n'est pas de décrire l'intégralité de la librairie AWT mais de présenter au lecteur un certain nombre d'exemples afin de lui donner une bonne vision de l'utilisation de ces outils, ainsi qu'une certaine méthodologie. Pour cela, nous reprenons l'exemple du sapin de Noël décoré, décrit à la section « Les tableaux à deux dimensions » du chapitre 9, « Collectionner un nombre fixe d'objets ». Cette fois, le sapin n'est plus affiché à l'aide de simples caractères mais avec des composants graphiques utilisant les méthodes prédéfinies de lalibrairie AWT.

Les fenêtres

L'affichage d'un outil graphique quel qu'il soit (bouton, menu, etc.) est toujours réalisé dans une **fenêtre**. Toute application graphique s'exécute à l'intérieur d'une zone délimitée, appelée fenêtre principale, dans laquelle sont placés barres d'outils, menus et zones de texte ou de dessin.

Cette fenêtre délimite le cadre d'exécution du programme, et tout élément se situant en dehors de la fenêtre fait partie d'une autre application. La fenêtre possède un bord et une barre de titre, dans laquelle se situent des boutons de fermeture et de mise en icône ou d'agrandissement, comme illustré à la Figure 11-1. Elle peut être déplacée ou agrandie sans que le programmeur ait à gérer lui-même ces actions.

En langage Java, la fenêtre principale est définie grâce à la classe `Frame`. Observez le programme suivant, qui décrit comment définir et afficher une `Frame`.

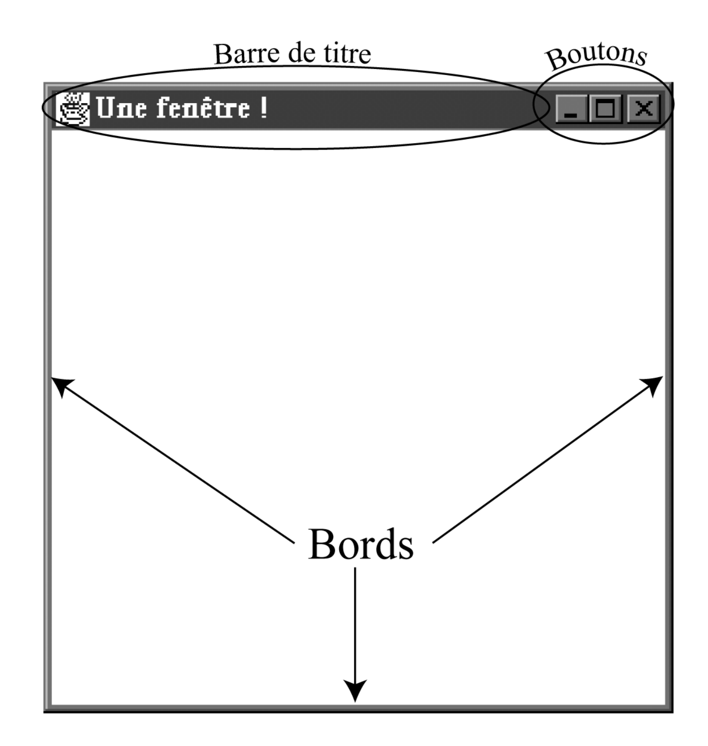


Figure 11-1.

La fenêtre principale délimite le lieu d'exécution du programme. Elle est constituée d'une bordure et d'une barre de titre.

Exemple : Une fenêtre

```
import java.awt.*;
class Fenetre {
 public    final static int HT = 300;
 public     final static int LG = 300;
 public static void main(String [] arg) {
    Frame F = new Frame();
    F.setTitle("Une fenetre!");        // met le titre
    F.setSize(HT, LG);                 // taille de la fenêtre
    F.setBackground(Color.gray);
    F.show();                          // affiche la fenêtre
 }
}
```

Nous constatons tout d'abord que la toute première instruction d'un programme qui utilise des objets graphiques est obligatoirement une instruction d'import du package de la librairie AWT (`import java.awt.* ;`). En effet, comme pour les vecteurs et les dictionnaires, les outils de la librairie graphique ne sont pas directement connus du compilateur.

Après avoir défini deux constantes, `HT` et `LG`, pour la hauteur et la largeur de la fenêtre, la fonction `main()` déclare et construit un objet `F` de type `Frame`. (`Frame F = new Frame();`). Comme toute classe, la classe `Frame` propose un ensemble de méthodes qui permettent la transformation de ses caractéristiques, notamment les suivantes :

- `setTitle()`, qui place la chaîne de caractères spécifiée en paramètre dans la barre de titre de la fenêtre.
- `setSize()`, qui définit la hauteur et la largeur de la fenêtre.
- `setBackground()`, qui donne une couleur de fond à la fenêtre.

Cela fait, la fenêtre est définie en mémoire mais n'est pas encore affichée à l'écran. Pour réaliser cet affichage, la méthode `show()`, définie par la classe `Frame`, est appliquée à l'objet `F`.

Pour connaître en détail l'ensemble des fonctionnalités de la classe `Frame`, reportez-vous au fichier `C:\jdk1.3\docs\api\java\awt\Frame.html`, après installation du JDK et de sa documentation.

Exemple : Résultat de l'exécution

Lors de l'exécution de ce programme, la fenêtre ayant pour titre « `Une fenetre !` » apparaît à l'écran, comme illustré à la Figure 11-1.

Le dessin

Une fois affichée, la fenêtre n'est pas encore directement fonctionnelle, et il n'est pas possible d'y afficher un dessin ou d'y écrire un texte. Il n'est pas non plus possible de fermer la fenêtre en cliquant sur le bouton de fermeture situé dans la barre de titre.

En effet, l'affichage d'un dessin ne peut être réalisé que par l'intermédiaire d'un objet de type **`Canvas`**.

En outre, pour fermer la fenêtre d'un simple clic sur le bouton approprié, le programme doit être capable d'entendre les clics de la souris. Nous étudions ce concept à la section « Les événements », en fin de chapitre.

Exemple : Dessiner un sapin de Noël

L'objectif de cet exemple est de réaliser l'affichage d'un sapin décoré en mode graphique. Fidèle à la méthode de travail qui consiste à découper un problème en plusieurs tâches indépendantes, nous allons réaliser l'affichage du sapin de Noël étape par étape.

Prenons pour hypothèse qu'un sapin est formé de triangles, disposés à l'écran de façon que leur juxtaposition réalise une forme de sapin. Nous placerons ensuite la décoration du sapin en modifiant la couleur de certains triangles.

Nous devons concevoir, dans un premier temps, un programme qui dessine un simple triangle de couleur verte.

Dessiner un triangle

Pour cela, nous définissons une classe `Dessin` qui hérite de la classe `Canvas` (`class Dessin extends Canvas`) . De cette façon, un objet de type `Dessin` correspond à une zone d'affichage où il est possible de dessiner des formes géométriques (point, droite, rectangle, etc.). Examinons attentivement cette classe :

```
import java.awt.*;
public class Dessin extends Canvas {
 public Dessin()    {
    setBackground(Color.white);
    setForeground(Color.green);
    setCursor(new Cursor(Cursor.CROSSHAIR_CURSOR));
 }
 public void paint (Graphics g)  {
    new Triangle(g);
 }
}
```

La classe `Dessin` est composée des deux méthodes suivantes :

- Le constructeur `Dessin()`, qui initialise une partie des caractéristiques d'un objet `Canvas`, à savoir :
  - La couleur de fond. La méthode `setBackground(Color.white)` place la couleur blanche en fond de la zone de dessin (`Canvas`).
  - La couleur d'avant-plan. La méthode `setForeground(Color.green)` assigne la couleur verte aux formes géométriques dessinées dans la zone de dessin.
  - Le curseur. La méthode `setCursor(new Cursor(Cursor.CROSSHAIR_CURSOR))` affiche un curseur en forme de croix lorsque le curseur de la souris se situe dans la zone de dessin.
- La méthode `paint()`, qui est une méthode prédéfinie de la classe `Canvas`. Cette méthode est appelée par l'interpréteur dès qu'il lui est nécessaire d'afficher un objet graphique. Elle est appelée lors de l'affichage de la fenêtre principale ou lorsque cette dernière réapparaît, après avoir été partiellement ou totalement cachée par une autre fenêtre.

La méthode `paint()` utilise en paramètre un objet `g` de type `Graphics` de façon à d'obtenir des informations sur le contexte graphique défini par l'application. Le contexte graphique est l'ensemble des informations utiles à l'affichage d'un objet. La couleur et la forme des caractères (fonte), par exemple, font partie du contexte graphique.

Ainsi, lorsque l'interpréteur affiche une fenêtre, il transmet à la méthode `paint()`, par l'intermédiaire du paramètre `g`, toutes les caractéristiques de l'affichage. En particulier, il lui transmet sa couleur d'avant-plan, initialisée à `color.green` dans le constructeur `Dessin()`.

Pour finir, l'exécution de la méthode `paint()` réalise l'affichage du triangle grâce à l'appel du constructeur de la classe `Triangle` (`new Triangle(g)`), dont voici la description :

```
import java.awt.*;
public class Triangle {
 private int centreX = Fenetre.LG/2;
 private int centreY = Fenetre.HT/2;
 private int [] xPoints = {centreX, centreX + 10, centreX - 10};
```

```
  private int [] yPoints = {centreY - 10, centreY + 10, centreY + 10};
  int nPoints = 3;
  public Triangle(Graphics g) {
     g.fillPolygon(xPoints, yPoints, nPoints);
  }
}
```

Les données de la classe `Triangle` correspondent à deux tableaux d'entiers définissant les sommets d'un triangle centré, comme illustré à la Figure 11-2.

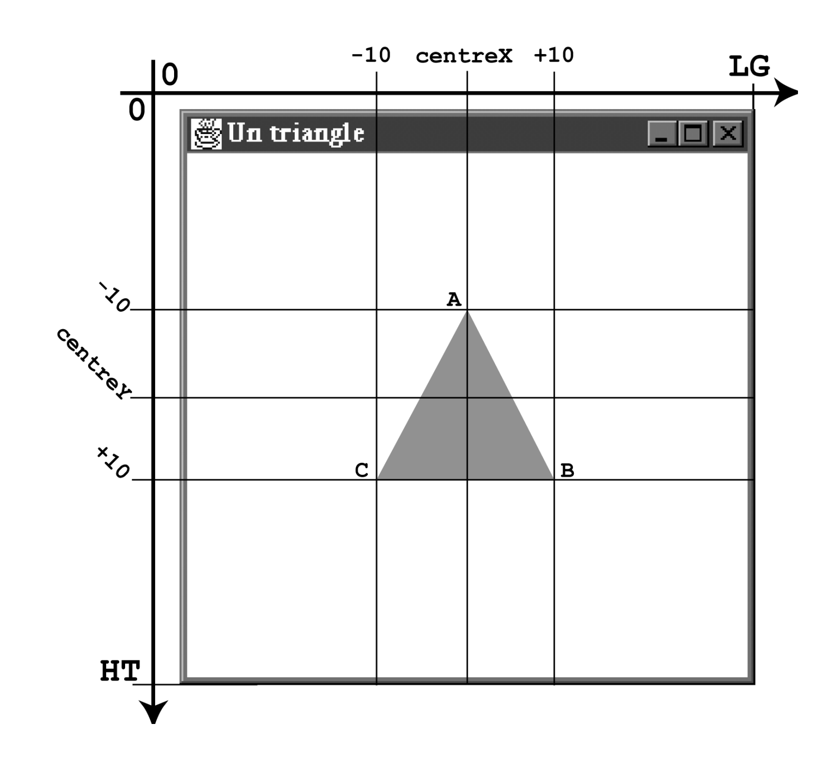


Figure 11-2.
Le triangle est construit à partir des sommets `A`, `B` *et* `C`*, dont les coordonnées sont calculées par rapport à l'origine de la zone de dessin située en haut et à gauche.*

Le sommet `A` est défini par le couple de coordonnées (`xPoints[0], yPoints[0]`), le sommet `B` par (`xPoints[1], yPoints[1]`) et `C` par (`xPoints[2], yPoints[2]`). Les tableaux définissent ainsi les sommets d'un polygone (triangle), centré par rapport à la fenêtre principale de l'application. Notez que les coordonnées des sommets sont définies par rapport à l'origine de l'objet `Canvas`, laquelle est située par défaut dans le coin supérieur gauche de cet objet.

L'affichage du triangle est réalisé grâce à la méthode `fillPolygon(xPoints, yPoints, nPoints)` , qui remplit de couleur le polygone spécifié en paramètre. La couleur de remplissage est déterminée par l'intermédiaire de l'objet `g` sur lequel la méthode est appliquée.

L'application `Fenetre`

Sans modification de l'application `Fenetre`, telle que définie à la section précédente de ce chapitre (*voir « Les fenêtres »*), aucun triangle n'apparaît. En effet, avant d'exécuter le programme, nous devons ajouter à la fenêtre le composant `Dessin`, de sorte que l'application puisse associer l'objet de type `Canvas` à la `Frame F`.

Pour cela, il suffit d'ajouter l'instruction `F.add(new dessin())`, comme l'illustre l'extrait de programme suivant :

```
class Fenetre {
//…
    Frame F = new Frame();
    F.setTitle("Un triangle");
//…
    F.add(new Dessin());
    F.show();              // affiche la fenêtre
 }
}
```

La méthode `add()` ajoute un composant graphique à la fenêtre principale. Ce composant est défini par la classe `Dessin`. Une fois `F` affichée, grâce à la méthode `show`, la méthode `paint()` est exécutée automatiquement, et le triangle s'affiche.

La construction du sapin

Sachant maintenant afficher un simple triangle, nous pouvons construire le sapin par juxtaposition d'un ensemble de triangles. Pour réaliser le bon positionnement des triangles, utilisons la technique développée à la section « Les tableaux à deux dimensions » du Chapitre 9, « Collectionner un nombre fixe d'objets ». Analysons la classe `Arbre`, qui reprend ce procédé :

```
import java.awt.*;
class Arbre {
 private int [][] sapin ;
 private Color décoration;
 public Arbre(int nl, Color c) {
    int nc = 2*nl-1;
    décoration = c;
    sapin = new int[nl][nc];
    int milieu = sapin[0].length/2;
    for ( int j = 0 ; j < nl ; j++)
        for ( int i = -j; i <= j; i++)
            sapin[j][milieu+i] = (int ) (5*Math.random()+1);
 }
 public void dessine(Graphics g) {
    Color Vert = Color.green;
    for (int i = 0; i < sapin.length; i++) {
        for (int j = 0; j < sapin[0].length; j++) {
            switch(sapin[i][j]) {
                case 1 :    new Triangle(i, j, g, décoration);
                            break;
                case 2 :    Vert = Vert.brighter();
                            new Triangle(i, j, g, Vert);
                            break;
                case 3 :    Vert = Vert.darker();
                            new Triangle(i, j, g, Vert);
                            break;
```

```
                    case 4 :    Vert = Vert.brighter();
                                new Triangle(i, j, g, Vert);
                                break;
                    case 5 :    Vert = Vert.darker();
                                new Triangle(i, j, g, Vert);
                                break;
                    case 6 :    Vert = Vert.brighter();
                                new Triangle(i, j, g, Vert);
                                break;
                }
            }
        }
    }
}
```

La classe `Arbre` est composée de deux données, le tableau d'entiers à deux dimensions `sapin` et la couleur `décoration`. Elle comporte en outre les deux méthodes suivantes :

- Le constructeur `Arbre()`, qui initialise la couleur de la `décoration`, et le tableau `sapin`, qui utilise la même technique que le sapin affiché en caractères graphiques. Les paramètres du constructeur rendent possible la création de sapins de taille et de couleur différentes.
- La méthode `dessine()` qui, en parcourant le tableau `sapin`, crée un triangle à l'aide du constructeur de la classe `Triangle` pour toute valeur `sapin[i][j]` différente de `0`. Grâce aux paramètres du nouveau constructeur `Triangle()`, le triangle est affiché à l'écran en fonction de sa position dans le tableau (indices `i` et `j`) et de la valeur du tableau (`sapin[i][j]`). Remarquez les méthodes `darker()` et `brighter()`, qui permettent de foncer ou d'éclaircir la couleur sur laquelle la méthode est appliquée. Grâce à ces méthodes, le sapin n'apparaît pas d'un vert uniforme.

Les paramètres du constructeur de la classe `Triangle` sont donc modifiés de façon à ne plus afficher un seul triangle vert au centre de la fenêtre mais un triangle d'une couleur et d'une position données. Pour réaliser cela, le constructeur est défini avec un ensemble de paramètres caractérisant la position en `x` et `y` à l'écran, ainsi que la couleur d'affichage du triangle. Examinons cette modification dans la classe `Triangle` ci-dessous :

```
import java.awt.*;
public class Triangle {
 private int pX = Fenetre.LG/2-50;
 private int pY = Fenetre.HT/2-50;
 private int [] xPoints = {0, 10, -10};
 private int [] yPoints = {-10, 10, 10};
 private int nPoints = 3;
 public Triangle(int col, int lig, Graphics g, Color c) {
    for (int i = 0; i < nPoints; i++) {
        xPoints[i] = xPoints[i]+(5*lig) + pX;
        yPoints[i] = yPoints[i]+(15*col) + pY;
    }
    g.setColor(c);
    g.fillPolygon(xPoints, yPoints, nPoints);
 }
}
```

Chaque triangle affiché ne se trouve plus au centre de la fenêtre mais à une position spécifiée en paramètre. Cette position est déterminée par les éléments suivants :

- un point de référence (`pX, pY`) défini en fonction de la taille de la fenêtre ;
- la position (`i, j`) du triangle dans le tableau `sapin`.

Ces valeurs sont transmises au constructeur grâce aux paramètres `col` et `lig`. Ces valeurs étant connues, les sommets du polygone sont calculés de façon à afficher ce dernier au bon endroit à l'écran. Comme tous les triangles prennent un certain espace en hauteur et en largeur, il est nécessaire d'appliquer un coefficient (`5` et `10`) aux indices `lig` et `col` pour que chaque triangle ne se superpose pas trop à ses voisins.

Le dessin du sapin

Pour que le sapin construit en mémoire s'affiche, ce dernier doit être placé dans la fenêtre de dessin. C'est ce que réalise la classe `Dessin` suivante :

```
import java.awt.*;
public class Dessin extends Canvas {
 private Color couleur = Color.green;
 public final static Color couleurFond = Color.white;
 private Arbre A;
 public Dessin()    {
    setBackground(couleurFond);
    setForeground(couleur);
    setCursor(new Cursor(Cursor.CROSSHAIR_CURSOR));
    A = new Arbre(8, Color.yellow);
 }
 public void paint (Graphics g)  {
    A.dessine(g);
 }
}
```

Cette classe reprend en grande partie l'architecture de la classe `Dessin`, décrite à la section « Dessiner un triangle », au début de ce chapitre. Pour remplacer le triangle par un sapin, le constructeur `Dessin()` crée en mémoire un objet `A` de type `Arbre`. La méthode `paint()` appelle ensuite la méthode `dessine()` par l'intermédiaire de l'objet `A` pour l'afficher à l'écran.

Remarquez que l'application `Fenêtre` n'est pas à modifier. Lorsque l'application est exécutée, une fenêtre s'affiche avec son composant de type `Dessin`. Ce dernier crée en mémoire un objet `A` de type `Arbre`, puis la méthode `paint()` est automatiquement appelée par l'interpréteur. Le sapin est alors affiché.

Les éléments de communication graphique

Outre les composants d'affichage tels que les `Frame` et les `Canvas`, la librairie AWT propose des outils de communication graphique, comme les boutons et les menus.

Ces outils offrent la possibilité d'écrire des applications munies d'une **interface graphique** réellement interactive. L'utilisateur manipule directement les objets proposés par l'interface, et cette dernière réagit en fonction des actions de l'utilisateur. Puisqu'il

n'est pas possible de savoir à l'avance quel objet va être manipulé, chaque composant doit être programmé de façon à réagir directement aux manipulations de l'utilisateur. Chaque manipulation est considérée comme un **événement**, auquel est associé un traitement, c'est-à-dire une **action**.

Afin d'étudier ces différents concepts, nous allons améliorer l'application du sapin de Noël en y insérant deux boutons : un premier bouton pour afficher un sapin avec de nouvelles décorations et un second pour quitter l'application.

Les boutons

Les boutons sont les composants de communication les plus utilisés pour créer des interfaces graphiques. Grâce à eux, par un simple clic, l'utilisateur valide son souhait de voir réaliser le traitement proposé par le bouton.

Les boutons sont définis dans la librairie AWT par la classe `Button`. Pour afficher un bouton, il suffit de l'ajouter à une fenêtre, comme nous l'avons déjà fait pour dessiner un objet `Canvas`. Examinons la classe `Fenetre`, dans laquelle nous allons insérer deux boutons :

```
import java.awt.*;
class Fenetre {
public    final static int HT = 300;
public     final static int LG = 300;
 public static void main(String [] arg) {
    Frame F = new Frame();
    // …
    F.add(new Dessin());
    F.add(new Button("Nouveau"));
    F.add(new Button("Quitter"));
    F.show();
 }
}
```

Dans cet exemple, deux boutons, portant les noms de « `Quitter` » et « `Nouveau` », sont ajoutés à la fenêtre `F` grâce à la méthode `add()`. Lorsque le programme est exécuté, l'affichage résultant est celui illustré à la Figure 11-3.



Figure 11-3.

Les composants graphiques s'affichent en se superposant les uns aux autres. C'est pourquoi le dernier bouton cache les autres composants.

Chaque composant (`Canvas` et `Button`) est ajouté à la fenêtre, sans que soient spécifiés ni sa position, ni sa taille. Dans cette situation, l'interpréteur affiche les composants en les superposant dans leur ordre d'arrivée. Le dernier bouton, « `Quitter` », cache par conséquent le composant `Dessin` ainsi que le bouton `Nouveau`.

Les conteneurs

Pour corriger cette erreur, il convient, lorsque vous souhaitez afficher plusieurs composants graphiques, de placer ces derniers à l'intérieur d'un conteneur (en anglais *container*).

Un conteneur est une sorte de boîte, qui contient tous les éléments de communication utilisés dans l'application. La plus part des boîtes à outils proposées dans les logiciels récents sont des conteneurs. Remarquez que seules les `Frame` ne peuvent être placées dans un conteneur.

Un conteneur est défini par la classe `Panel` du package `java.awt`. Examinons comment l'utiliser dans le programme suivant :

```
import java.awt.*;
public class DesBoutons extends Panel {
 public DesBoutons() {
  // initialisation
  setBackground(Color.lightGray);
  // Les boutons
  Button bNouveau = new Button ("Nouveau");
  this.add(bNouveau);
  Button bQuitter = new Button ("Quitter");
  this.add(bQuitter);
}
```

La classe `DesBoutons` est définie comme classe héritant de la classe `Panel` (`DesBoutons extends Panel`) . Elle est composée d'un constructeur, qui crée en mémoire deux boutons, `bNouveau` et `bQuitter`, et les ajoute ensuite au conteneur grâce à la méthode `add()`. Par défaut, les boutons sont affichés au centre du `Panel` par ordre d'arrivée.

Remarquez l'application du terme `this` aux méthodes `add()`. Facultatif, ce terme indique à l'interpréteur qu'il doit ajouter ces objets (les boutons) à l'objet qu'il est en train de construire, c'est-à-dire au `Panel` nommé `DesBoutons`. L'expression `this` représente l'objet qui se construit en mémoire.

Une fois défini, le conteneur doit être ajouté à la fenêtre. Pour éviter toute superposition du conteneur à l'objet `Canvas`, il est possible d'indiquer à l'interpréteur comment afficher les éléments les uns par rapport aux autres. Utilisons à cette fin les termes `"South"`, `"North"`, `"Center"`, `"East"` et `"West"` en paramètre de la méthode `add()`.

Le programme `Fenetre` ci-dessous utilise cette technique pour afficher correctement les deux boutons :

```
import java.awt.*;
class Fenetre{
//..
 public static void main(String [] arg) {
    Frame F = new Frame();
//..
```

```
      F.add(new Dessin(), "Center");
      F.add(new DesBoutons(), "South");
      F.show();
   }
 }
```

Grâce aux paramètres "Center" et "South", les composants s'affichent correctement la fenêtre de dessin, au-dessus de la boite à boutons, comme illustré à la Figure 11-4.

Notre application possède maintenant deux boutons. Pourtant, lorsque l'utilisateur clique sur l'un ou l'autre de ces boutons, rien ne se passe : l'affichage de nouveaux sapins n'est pas effectué, et il n'est pas non plus possible de quitter l'application en cliquant sur le bouton "Quitter".

C'est qu'il ne suffit pas d'afficher un bouton avec un texte correspondant à l'action souhaitée pour voir cette action se réaliser. La classe Button ne fait que définir les attributs graphiques des boutons. Pour associer un bouton à une action, il faut encore gérer les événements.



Figure 11-4.
Une fois la position des composants définie par rapport aux bords de la fenêtre principale, chaque composant s'affiche correctement.

Les événements

En langage Java, la gestion des événements est réalisée par l'intermédiaire d'objets spécifiques, appelés écouteurs (en anglais *listener*). De façon simplifiée, on peut dire que, lorsque l'utilisateur clique sur un bouton ou sur une commande de menu, le composant concerné émet un événement à l'attention de l'écouteur.

Le traitement de cet événement est réalisé par l'écouteur d'événement (Eventlistener) et non pas par le composant lui-même. Le langage Java gère les événements en suivant un modèle dit « par délégation » (en anglais *delegation model*), le traitement de l'événement étant délégué à un autre composant que celui qui l'a perçu.

Les types d'événements

Chaque composant graphique émet un événement propre à sa classe, et il existe donc plusieurs types d'événements. On distingue les Événements de bas niveau et les Événements de haut niveau.

Événements de bas niveau

Les événements de bas niveau sont les événements créés à partir de la souris, du clavier ou d'une fenêtre. Le tableau suivant résume les types d'événements de bas niveau les plus utilisés.

Écouteur	Comportement à programmer	
`MouseListener` Écoute les événements liés à la souris.	`mousePressed(MouseEvent)`	Appelé lors d'une pression sur un bouton de la souris.
	`mouseReleased(MouseEvent)`	Appelé lorsqu'un bouton de la souris est relâché.
	`mouseExited(MouseEvent)`	Appelé lorsque la souris sort de la fenêtre.
	`mouseEntered(MouseEvent)`	Appelé lorsque la souris entre dans la fenêtre.
	`mouseClicked(MouseEvent)`	Appelé lors d'un simple clic de souris.
MouseMotionListener Écoute les événements liés à la souris lorsqu'elle se déplace.	`mouseDragged(MouseEvent)`	Appelé lorsque la souris est déplacée, bouton enfoncé.
	`mouseMoved(MouseEvent)`	Appelé lorsque la souris est déplacée, bouton relâché.
WindowListener Écoute les événements liés à la fenêtre.	`windowClosing(WindowEvent)`	Appelé lorsque la fenêtre est en train de se fermer.
	`windowClosed(WindowEvent)`	Appelé lorsque la fenêtre est fermée.
	`windowOpened(WindowEvent)`	Appelé lors de l'ouverture de la fenêtre.
	`windowIconified(WindowEvent)`	Appelé lorsque la fenêtre est mise sous forme d'icône.
	`windowDeiconified(WindowEvent)`	Appelé lorsque l'icône est agrandie à la taille de la fenêtre.
	`windowActived(WindowEvent)`	Appelé lorsque la fenêtre est activée et reçoit les événements du clavier.
	`windowDeactived(WindowEvent)`	Appelé lorsque la fenêtre est désactivée et perd les événements du clavier.

Pour gérer un événement lié à la souris, par exemple, il convient de définir un écouteur de type `MouseListener` et de décrire le comportement de l'application pour chaque méthode associée à cet écouteur.

Événements de haut niveau

Les événements de haut niveau sont liés, non plus aux comportements du composant graphique, mais à ses actions possibles. Ainsi, un clic de souris sur un bouton ne génère plus un événement spécifique du composant mais un comportement défini par le programmeur.

Écouteur	Comportement à programmer	
ActionListener Écoute les événements d'action.	`actionPerformed(ActionEvent)`	Appelé lorsqu'une action est émise.

Ainsi, une action est associée à un bouton en définissant un écouteur d'action, qui, par l'intermédiaire de la méthode `actionPerformed`, réalise l'action souhaitée.

Exemple : Associer un bouton à une action

Lorsque l'utilisateur clique sur les boutons `"Nouveau"` ou `"Quitter"` de l'application développée dans ce chapitre, il souhaite voir se réaliser deux actions distinctes : soit l'affichage d'un nouveau sapin, soit la fermeture de la fenêtre.

Chaque clic de souris sur un bouton est associé à une action spécifique, qui utilise un événement de haut niveau. Par conséquent, chaque bouton doit être muni d'un écouteur d'action. Cela est réalisé dans la classe `DesBoutons`, comme ci-dessous :

```
import java.awt.*;
import java.awt.event.*;
public class DesBoutons extends Panel {
 public DesBoutons(Dessin d)  {
  //…
    Button bNouveau = new Button ("Nouveau");
    Button bQuitter = new Button ("Quitter");
    bNouveau.addActionListener(new GestionAction(1, d));
    this.add(bNouveau);
    bQuitter.addActionListener(new GestionAction(2, d));
    this.add(bQuitter);
 }
}
```

Remarquez l'instruction d'import (`import java.awt.event.*;`), qui indique au compilateur le package où sont définies les méthodes de gestion des événements utilisées dans la classe.

La mise en place des écouteurs d'actions est réalisée grâce à la méthode `addActionListener()`.

Le constructeur `GestionAction()`, placé en paramètre de la méthode `addActionListener()`, permet de préciser quel comportement doit adopter l'application en fonction du bouton qui émet l'événement. En effet, les paramètres de ce constructeur transmettent à la fois une valeur entière différente (`1` ou `2`) suivant le bouton émetteur (`bNouveau` ou `bQuitter`) et l'adresse de l'objet (`d`) sur lequel est dessiné le sapin. Examinons comment sont gérés ces paramètres dans la classe `GestionAction` :

```
import java.awt.*;
import java.awt.event.*;
public class GestionAction implements ActionListener {
 private int n;
 private Dessin d;
```

```
    public GestionAction( int n, Dessin d) {
       this.n = n;
       this.d = d;
    }
    public void actionPerformed(ActionEvent e){
       switch (n) {
           case 1 :    d.nouveau();
                       break;
           case 2 :    System.exit(0);
                       break;
       }
    }
}
```

La classe `GestionAction` fait appel à plusieurs notions importantes, développées dans les sections qui suivent.

Le terme `implements`

La classe `GestionAction` implémente la classe `ActionListener` (`GestionAction implements ActionListener`) ; elle n'en hérite pas.

En réalité, la classe `ActionListener`, comme tous les autres `listener`, n'est pas véritablement une classe. Il s'agit en fait d'une classe **abstraite**.

Les méthodes définies par un `listener` ne peuvent pas être prédéfinies par le langage Java. Une action, c'est-à-dire un comportement associé à un bouton, ne peut être décrite que par le programmeur, selon la façon dont il conçoit son application. On dit alors que la classe `ActionListener`, ainsi que tous les autres `listener`, est une **interface** qui définit simplement les différents modes de comportement.

Lorsqu'une classe dérive d'une interface, le terme `implements` doit être utilisé au lieu du terme `extends`.

De plus, lorsqu'une classe implémente une interface, le compilateur Java exige de décrire l'intégralité des méthodes définies par l'interface. En effet, dans le cas où l'une des méthodes n'est pas définie, le compilateur indique une erreur en précisant le nom de la méthode manquante.

Dans notre exemple, l'interface `ActionListener` ne définit qu'une seule méthode (`actionPerformed()`). Nous n'avons donc aucune difficulté à décrire l'intégralité des méthodes proposées par cette interface.

Le terme `this`

La classe `GestionAction` possède deux variables d'instance, `n` et `d`, de façon à mémoriser la valeur respective transmise par les boutons, ainsi que l'adresse de la zone graphique où le sapin est dessiné. Ces valeurs sont initialisées par l'intermédiaire des paramètres du constructeur `GestionAction()`, qui sont également nommés `n` et `d`.

Pour éviter toute confusion entre les données de la classe et les paramètres du constructeur, il est nécessaire d'employer le terme `this`. Ce terme, appliqué à `n` et `d`, précise au compilateur qu'il s'agit des variables de l'instance qui se construit. Sans `this`, les mêmes noms de variables correspondraient aux paramètres du constructeur.

La méthode actionPerformed()

Une fois les données initialisées, la méthode actionPerformed() est automatiquement exécutée par l'interpréteur, lorsqu'une action est émise par l'un des boutons suite à un clic de l'utilisateur. Cette dernière réalise alors, suivant la valeur transmise au constructeur (1 ou 2), soit la sortie du programme, soit l'affichage d'un nouveau sapin, grâce à la méthode nouveau() (à insérer dans la classe Dessin) décrite ci-dessous :

```
public void nouveau()  {
    A = new Arbre(6, Color.red);
    repaint();
}
```

À l'exécution de cette méthode, l'objet A est recalculé à l'aide du constructeur Arbre(). Ensuite, la méthode repaint() efface automatiquement la zone d'affichage d sur laquelle la méthode est appliquée et appelle la méthode paint() définie dans la classe Dessin.

La donnée Dessin d

Le bouton bNouveau a une incidence sur la zone de dessin puisqu'un nouveau sapin doit être affiché dans cette zone suite à un clic sur le bouton. Cet effet est réalisé par le biais de la méthode nouveau(), appliquée à l'objet d de type Dessin dans la méthode actionPerformed(). L'objet d est déclaré comme variable d'instance de la classe GestionAction. Il contient l'adresse de la zone d'affichage créée dans l'application Fenetre, comme l'illustre l'extrait de programme suivant :

```
public class Fenetre {
//…
 public static void main(String [] arg) {
    Frame F = new Frame();
    //…
    Dessin page = new Dessin() ;
    F.add(page, "Center");
  F.add(new DesBoutons(page), "South");
} }
```

La construction de l'objet page a pour résultat de stocker en mémoire l'adresse du composant graphique de type Canvas. Une fois cette adresse transmise au constructeur de la classe DesBoutons, ce dernier peut transmettre à son tour l'adresse de l'objet page au constructeur de la classe GestionAction par l'intermédiaire du paramètre formel d de type Dessin. Grâce à la transmission de l'adresse de la zone graphique en paramètre des constructeurs, les nouveaux sapins s'affichent dans le composant graphique approprié.

Exemple : Fermer une fenêtre

Pour fermer une fenêtre en cliquant directement sur l'icône de fermeture de la fenêtre située dans la barre de titre, l'événement « clic sur le bouton de fermeture de la fenêtre » doit être associé à l'instruction qui permet de stopper l'exécution du programme. Comme le précise le tableau de description des écouteurs, l'événement « clic sur le bouton de fermeture de la fenêtre » est un événement de bas niveau, géré par l'écouteur WindowListener.

En effet, lorsque l'utilisateur ferme la fenêtre par l'intermédiaire de l'icône appropriée, ce dernier émet un événement de fermeture de fenêtre. En recevant cet événement, l'écouteur des événements de fenêtre (`WindowListener`) exécute automatiquement la méthode `windowClosing()`.

Par conséquent, le programme qui réalise la fermeture d'une fenêtre doit effectuer les deux opérations suivantes :

- placer un écouteur d'événement de fenêtre dans le composant graphique autorisé à être fermé de la sorte ;
- programmer la méthode `windowClosing()` en y insérant l'instruction `System.exit(0)` de façon à sortir de l'application.

Placer un écouteur d'événement de fenêtre

Le premier point est réalisé dans la classe `Fenetre` de la façon suivante :

```
import java.awt.*;
class Fenetre {
 //..
 public static void main(String [] arg) {
    Frame F = new Frame();
    //…
    F.addWindowListener(new GestionFenetre());
    F.show();
 }
}
```

Une fois la méthode `addWindowListener()` appliquée à la fenêtre `F`, cette dernière est à même d'écouter les événements émis par ses propres composants.

Programmer la méthode `windowClosing()`

Le second point est résolu grâce à l'appel du constructeur `GestionFenetre()` en paramètre de la méthode `addWindowListener()`, ce dernier définissant le comportement adopté en face de l'événement entendu. Examinons plus attentivement la classe `GestionFenetre` :

```
import java.awt.event.*;
public class GestionFenetre extends WindowAdapter{
 public void windowClosing(WindowEvent e){
 System.exit(0);
 }
}
```

Remarquez que la classe `GestionFenetre` hérite de la classe `WindowAdapter` au lieu d'implémenter l'interface `WindowListener`.

L'écouteur `WindowListener` possède sept comportements spécifiques *(voir précédemment le tableau des événements de bas niveau).* Si nous implémentons directement cette interface, comme nous l'avons fait pour `ActionListener`, nous sommes contraints par le compilateur Java à définir l'ensemble des sept comportements.

Or, pour fermer la fenêtre, un seul comportement est à retenir : celui défini par la méthode `windowClosing()`. En utilisant un « `Adapter` », plutôt qu'un « `Listener` », nous n'avons plus l'obligation de définir l'intégralité des sept comportements mais uniquement la ou les méthodes de votre choix. Pour notre exemple, seule la méthode `windowClosing()` nous intéresse. C'est pourquoi elle seule est décrite dans la classe `GestionFenetre`.

Ainsi, lorsque l'utilisateur clique sur l'icône de fermeture de la fenêtre, un événement de fermeture de fenêtre est émis. L'événement est capté et traité par l'écouteur `WindowListener`, qui exécute automatiquement la méthode `windowClosing()`. Celle-ci termine l'exécution de l'application grâce à l'instruction `System.exit(0)` .

Quelques principes

L'analyse des exemples précédents montre que la gestion d'un événement quel qu'il soit passe par les trois étapes suivantes :

- Déterminer le composant qui émet l'événement et lui associer un écouteur. Cette association est réalisée par une méthode ayant pour nom `addxxxListener()`, où `xxx` représente le composant émetteur (`Window`, `Mouse`, etc.).
- Créer une classe `gestionDelEvenement` qui implémente l'interface `xxxListener` (`implements`) ou dérive de la classe `xxxAdapter` (`extends`), selon que vous souhaitiez traiter tout ou partie des méthodes proposées par l'écouteur.
- Développer les méthodes souhaitées, c'est-à-dire décrire les instructions composant les méthodes définies par l'interface utilisée.

Les applets

Les applications développées dans cet ouvrage sont, jusqu'à présent, des programmes exécutés directement sur le système d'exploitation de la machine par l'interpréteur Java. Ces applications sont appelées applications autonomes.

Le langage Java offre la possibilité de créer des applications exécutables par l'intermédiaire d'un navigateur Web, tel que Netscape ou Internet Explorer.

Ces applications, appelées des `applets` , raccourci des termes *appl*ication et Intern*et*, sont exécutables sur le réseau Internet. Une applet ne peut s'exécuter qu'au travers d'un navigateur et n'est donc pas une application autonome.

Le support d'exécution d'un navigateur Web est la page HTML (*HyperText Markup Language*). C'est pourquoi une applet doit être insérée dans une page HTML pour être lue et exécutée.

Une page HTML

Une page HTML est un fichier texte, constitué de balises (mots clés) indiquant au navigateur la façon dont il doit afficher le contenu de la page (mise en page). L'exemple qui suit décrit le plus petit fichier utilisable pour exécuter une applet Java.

Exemple : L'applet default.htm

```
<HTML>
    <HEAD>
        <TITLE>Un sapin de Noel</TITLE>
    </HEAD>
    <BODY>
        <APPLET CODE="Fenetre" WIDTH=300 HEIGHT=300>
        </APPLET>
    </BODY>
</HTML>
```

Une page HTML est donc un fichier débutant toujours par la balise `<HTML>` et se terminant par sa balise inverse `</HTML>`. Elle est constituée des deux grands blocs suivants :

- **L'en-tête**, composé des balises `<HEAD>` et `</HEAD>`. Dans votre exemple, l'en-tête permet d'affecter un titre à la page, entre les balises `<TITLE>` et `</TITLE>`.
- **Le corps**, défini par les balises `<BODY>` et `</BODY>`, qui décrit le contenu de la page. L'insertion d'une applet est toujours effectuée dans le corps de la page. Il suffit d'utiliser pour cela la balise `<APLLET>`, composée des trois paramètres `CODE`, `WIDTH` et `HEIGHT`, qui indiquent au navigateur le nom du fichier de l'applet à exécuter, ainsi que sa largeur et sa hauteur.

Le fichier écrit est sauvegardé sous un nom ayant l'extension `.htm`. Par défaut, tous les navigateurs lisent les pages `HTML` portant le nom `default.htm`. Par conséquent, sauvegardons la page `HTML` décrite ci-dessus dans un fichier portant ce nom.

Construire une applet

Une applet diffère d'une application autonome par plusieurs aspects :

- La fonction `main()` disparaît pour être remplacée par la méthode `init()`.
- La classe où se situe la méthode `init()` hérite de la classe `Applet`.
- Le support d'affichage des composants graphiques n'est plus une `Frame`. Le navigateur se charge d'afficher tous les composants graphiques. Il possède également une barre de titre, des bords et des boutons. L'utilisation d'une `Frame` devient donc inutile.
- L'instruction `setTitle("Un sapin de Noel")` est également inutile puisque le titre de la fenêtre est maintenant géré par la page HTML. Il s'affiche dans la barre de titre du navigateur.

Exemple : Un sapin de Noël en applet

L'applet `Fenetre` s'écrit maintenant de la façon suivante :

```
import java.awt.*;
import java.applet.*;
public class Fenetre extends Applet {
 public    final static int HT = 300;
 public    final static int LG = 300;
 public void init()    {
    Dessin D;
    setSize(HT, LG);
```

```
        setBackground(Color.darkGray);
        setLayout(new BorderLayout());
        add(D = new Dessin(), "Center");
        add(new DesBoutons(D), "South");
    }
  }
```

Notez l'instruction d'import (`import java.applet.*;`) , qui indique au compilateur où se trouve le package définissant les applets.

L'applet est affichée par le navigateur, qui exécute en premier lieu la méthode `init()`, tout comme l'interpréteur Java exécute la fonction `main()`, dans le cas d'une application Java autonome.

Les instructions de la fonction `init()`sont exécutées ligne à ligne, et le résultat est équivalent dans la forme à l'application décrite précédemment. Remarquons cependant que :

- Les méthodes ne sont pas appliquées à un objet `F` de type `Frame` mais à l'objet qui est en train de se construire, c'est-à-dire à l'applet elle-même.
- Un `BorderLayout` est ajouté, par l'intermédiaire de la méthode `setLayout()`. La mise en page (en anglais *layout*) correspond à une stratégie d'affichage des composants graphiques. Par exemple, pour un `Panel` ou une `Applet`, la stratégie par défaut est le `FlowLayout`, c'est-à-dire l'affichage centré des composants dans leur ordre d'arrivée.

Pour une `Frame`, la stratégie d'affichage est le `BorderLayout`, qui permet l'affichage des objets par rapport aux bords de l'objet. Ces bords sont nommés `East`, `North`, `West` et `South`. L'ajout d'un composant dans un objet dont la stratégie d'affichage est le `BorderLayout` est réalisé en indiquant en paramètre le bord dont il doit s'approcher le plus. C'est ce que nous avons réalisé, sans le savoir, dans l'application `Fenetre` de la section précédente.

L'applet par défaut ne travaille pas avec un `BorderLayout` mais avec un `FlowLayout`. Sans modification du gestionnaire de mise en page (`Layout Manager`), l'applet affiche les composants dans leur ordre d'arrivée, soit d'abord la fenêtre de dessin, puis la boîte à boutons en superposition, enfin le sapin disparaissant sous les boutons.

Grâce à la mise en place du `BorderLayout` (`setLayout(new BorderLayout())`), l'affichage est corrigé, et tous les éléments se trouvent à leur place, comme illustré à la Figure 11-5.

L'utilitaire AppletViewer

L'exécution d'une applet s'effectue en général en chargeant la page HTML correspondante dans un navigateur. L'affichage s'exécute automatiquement.

Une solution plus rapide consiste cependant à faire appel à l'utilitaire AppletViewer, livré avec le JDK. AppletViewer exécute une applet sans l'environnement du navigateur. Pour l'utiliser, il suffit d'écrire une ligne de commande dans une fenêtre de commandes MS-DOS (environnement Windows.) ou dans une fenêtre de script (environnement Unix.).

Cette commande est la suivante :

```
appletviewer default.htm
```

L'exécution de cette commande a pour résultat l'affichage de l'applet illustrée à la Figure 11-5.

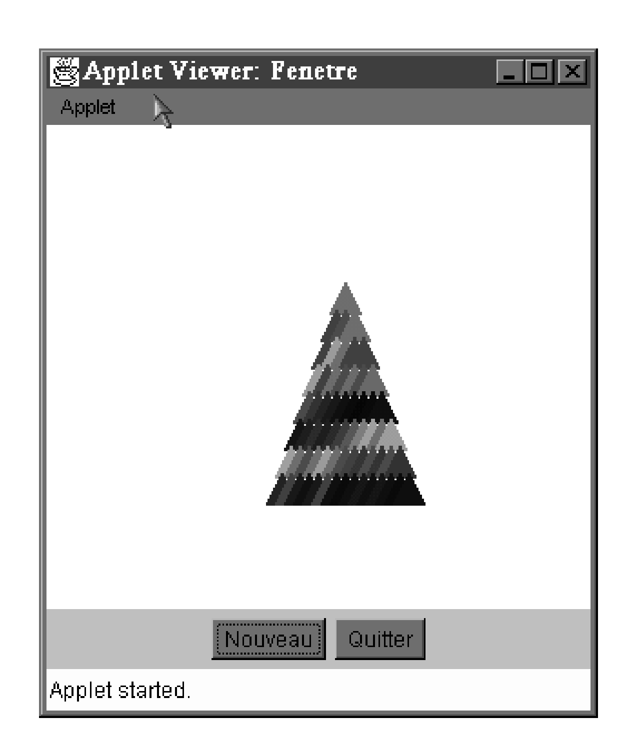


Figure 11-5.
Grâce à l'utilitaire AppletViewer, il est possible d'afficher une applet en dehors de tout navigateur Web.

Résumé

L'essentiel des composants graphiques développés par le langage Java est défini dans la librairie AWT *(Abstract Windowing Toolkit).*

Le support principal d'affichage d'une application graphique est la **`Frame`**. Cette dernière est composée des éléments suivants :

- une barre de titre possédant des boutons pour la fermeture, l'agrandissement et la mise en icône de la fenêtre ;
- des bords délimitant la zone d'exécution de l'application.

Pour dessiner ou afficher du texte dans une `Frame`, il convient d'utiliser des objets de type **`Canvas`** ou `TextArea`. Le contexte graphique définissant les attributs d'affichage, tels que la couleur, le type de fonte, etc., est géré par la classe `Graphics`.

Les interfaces graphiques sont construites à l'aide des éléments de communication graphique suivants :

- composants graphiques tels que **boutons** (`Button`) et menus ;
- **événements** associant, par exemple, un clic de souris sur un bouton à une action.

On distingue les événements de **haut niveau** (un clic est associé à une action) et les événements de **bas niveau** (un clic sur un composant émet un événement propre à ce composant).

Le langage Java propose en outre des composants graphiques spécifiques, appelés `Applet` (*appl*ication sur Intern*et*), définis dans le package `java.applet`.

Une applet n'est pas exécutée par l'interpréteur Java mais par le navigateur, à travers une page HTML *(HyperText Markup Language)*, qui est un fichier texte constitué de balises (mots clés) indiquant au navigateur comment il doit mettre en page les informations contenues dans le fichier.

Pour être exécutable, le programme définissant l'applet doit faire appel à la méthode `init()` en lieu et place de la fonction `main()`.

Exercices

L'objectif de cet exercice est d'améliorer le programme réalisé tout au long de ce chapitre. L'interface graphique à construire propose une case à cocher (en anglais *check box*) permettant de répondre aux conditions suivantes :

- Si la case `Taille fixe` est cochée, les nouveaux sapins de taille constante sont dessinés avec une guirlande formée de cercles de différentes couleurs.
- Si la case n'est pas cochée, les nouveaux sapins sont également dessinés mais avec une taille variable.

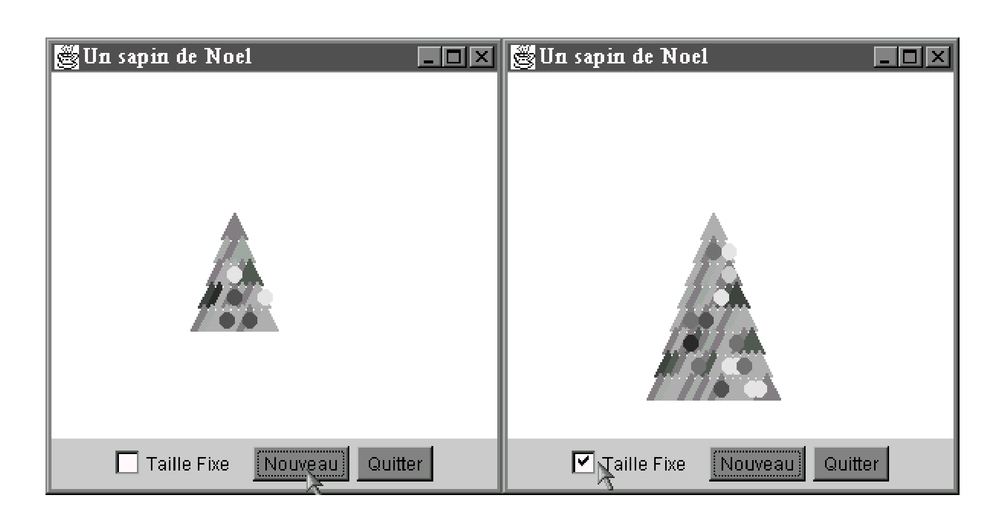


Figure 11-6. *L'application propose une case à cocher pour déterminer si la taille des nouveaux sapins doit être fixe ou non*

Pour construire cette application, nous vous proposons de suivre les différentes étapes décrites ci-dessous.

Comprendre les techniques d'affichage graphique

11.1 Pour afficher un sapin de taille différente chaque fois que l'utilisateur clique sur le bouton `Nouveau` :

a. Recherchez dans l'ensemble des classes de l'application `Fenetre`, la méthode associée au clic sur le bouton `Nouveau`.

b. Modifiez cette méthode de façon que l'arbre se construise avec une taille aléatoire, variant entre `3` et `10`, par exemple.

c. Une fois ces modifications réalisées, compilez l'application, et vérifiez le bon fonctionnement du bouton `Nouveau`.

11.2 Pour dessiner une guirlande de cercles de couleurs différentes :

a. Avant d'afficher une guirlande, modifiez les classes `Triangle` et `Arbre` de sorte que le sapin ne soit affiché qu'à l'aide de triangles verts. Vérifiez l'exécution du programme.

b. Pour afficher une guirlande de couleur rouge, créez une classe `Boule` en vous inspirant de la classe `Triangle`.

Notez que la méthode `fillOval(x, y, l, h)` permet l'affichage de cercles remplis. Elle s'applique à un objet `Graphics`, comme la méthode `fillPolygon()`. Les paramètres `x` et `y` représentent la position à l'écran du coin supérieur gauche du rectangle englobant le cercle, `l` et `h` représentant la largeur et la hauteur de ce même rectangle.

c. Modifiez ensuite la méthode `dessine()` de la classe `Arbre`, de façon à construire et à afficher par-dessus le sapin des objets `Boule` lorsque le tableau `sapin` vaut `1`.

Compilez et exécutez le programme afin de vérifier le bon affichage de la guirlande.

d. Pour afficher une guirlande de couleurs différentes, définissez dans la classe `Boule` un tableau de plusieurs couleurs, comme suit :

```
Color [] couleur = {Color.red, Color.blue, Color.yellow, Color.cyan,
➥Color.magenta};
```

Le choix de la couleur est ensuite effectué dans le constructeur de la classe `Boule` en tirant au hasard une valeur comprise entre `0` et `5`. Cette valeur est utilisée comme indice du tableau de couleurs pour initialiser la couleur d'affichage (`setColor()`) à la couleur du tableau cor respondant à l'indice tiré au hasard.

Apprendre à gérer les événements

11.3 Placer une case à cocher dans la boîte à boutons :

a. Sachant que la classe décrivant les cases à cocher a pour nom `Checkbox`, ajoutez un objet de ce type dans la boîte à boutons de l'application `Fenêtre`. Le texte (« `Taille fixe` ») suivant la case à cocher est placé en paramètre du constructeur de la classe.

b. L'écouteur d'événement associé aux objets de type `Checkbox` s'appelant `ItemListener`, ajoutez cet écouteur à la case à cocher.

11.4 Associer l'événement à l'action. Lorsque la case est cochée, les nouveaux sapins affichés par le bouton `Nouveau`, sont de taille fixe. Inversement, lorsque la case n'est pas cochée, les sapins sont de taille aléatoire. L'état de la case à cocher a donc une influence sur l'affichage du sapin géré par le bouton `Nouveau`. C'est pourquoi il est logique de gérer l'écouteur `ItemListener` dans la même classe qu'`ActionListener`.

a. Sachant que l'interface `ItemListener` ne définit qu'un seul comportement `itemStateChanged()`, modifiez l'en-tête de la classe `GestionAction` de façon qu'elle implémente les deux interfaces `ActionListener` et `ItemListener` en séparant les deux termes par une virgule.

b. Analysez la méthode `itemStateChanged()` décrite ci-dessous, et déterminez comment déclarer la variable `OK` pour qu'elle puisse être également visible de l'objet `bNouveau`.

```
public void itemStateChanged(ItemEvent e) {
    if(e.getStateChange() == ItemEvent.SELECTED)
        OK = false;
    else OK = true;
 }
```

c. Sachant que les sapins de taille aléatoire sont affichés par l'intermédiaire de la méthode `nouveau()` (classe `Dessin`), modifiez la méthode de façon que la taille du sapin soit fixe ou aléatoire, en fonction de la valeur de la variable `OK`.

Le projet « Gestion d'un compte bancaire »

L'objectif est de réaliser des statistiques sur les comptes bancaires enregistrés dans les fichiers créés au chapitre précédent. Le résultat de ces statistiques est affiché dans une fenêtre, comme illustré à la Figure 11-7.

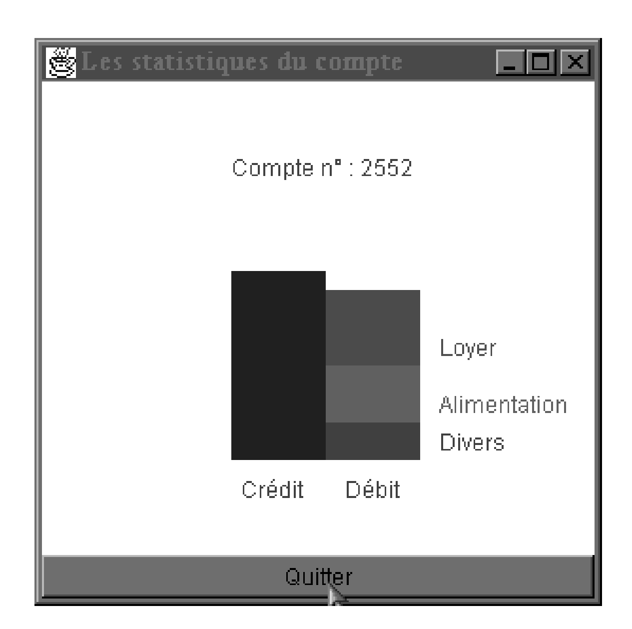


Figure 11-7.
Histogramme empilé du compte n° 2552.

Calcul de statistiques

Les classes `ListeCompte` et `Compte`

En reprenant la fonction `pourcentage()` réalisée en exercice à la fin du Chapitre 5, « De l'algorithme paramétré à l'écriture de fonctions », et sachant que l'objectif est de calculer en pourcentage les dépenses réalisées en fonction du motif de la dépense :

a. Déterminer toutes les données, définies dans les classes `Compte` et `LigneComptable`, nécessaires aux calculs des statistiques d'un compte.

b. Vérifier que ces données soient accessibles depuis l'extérieur de la classe. Si tel n'est pas le cas, écrire les méthodes d'accès en consultation pour chacune d'entre elles.

c. Après modifications, compiler et exécuter le programme de façon à créer un fichier de comptes (`Compte.dat`).

La méthode `statParMotif()`

a. En reprenant l'algorithme de calcul de statistiques proposé en exemple du Chapitre 1, « Stocker un information », écrire la méthode `statParMotif()` qui calcule le pourcentage des dépenses en fonction du motif enregistré.

b. Sachant que cette méthode est définie à l'intérieur d'une classe appelée `Stat`, déterminer les variables d'instance de cette classe.

c. Avant de passer à l'affichage graphique, écrire une application qui :

- lise le fichier « `Compte.dat` » créé à l'étape précédente ;
- utilise la méthode `statParMotif()` pour calculer et afficher à l'écran les statistiques d'un compte donné.

d. Vérifier la validité des calculs réalisés.

L'interface graphique

Pour calculer les statistiques d'un compte, l'utilisateur doit fournir le numéro du compte choisi. Après lecture du fichier `Compte.dat`, et connaissant ce numéro, l'application vérifie s'il existe en mémoire. Si tel est le cas, elle affiche dans une fenêtre le résultat sous forme d'histogrammes empilés. Dans le cas contraire, elle affiche un message indiquant que ce compte n'est pas connu et attend la saisie d'un nouveau numéro.

Deux étapes sont donc à réaliser, la saisie d'un numéro de compte et l'affichage de l'histogramme.

L'affichage de l'histogramme

Pour afficher l'histogramme empilé, il est nécessaire de connaître les pourcentages de dépenses en fonction des motifs déclarés. Ces valeurs sont calculées dans la classe `Stat`, construite précédemment.

a. En s'inspirant de la méthode `dessine()`, présentée en exemple au cours de ce chapitre, écrire dans la classe `Stat` la méthode `dessine()`, de façon à afficher :

- un premier rectangle de hauteur `100` et de largeur `50` représentant l'unité de crédit (`100`) ;
- des rectangles de couleur et de hauteur différentes suivant les pourcentages calculés par la méthode `statParMotif()`.

  Noter que l'affichage d'un rectangle rempli s'effectue par l'intermédiaire de la méthode `fillRect(x, y, l, h)`, où `x` et `y` représentent la position à l'écran du coin supérieur gauche du rectangle, et `l` et `h` sa largeur et sa hauteur. L'affichage d'un texte est réalisé par la méthode `drawString(texte, x, y)` , où `texte` est un objet de type `String` dans lequel sont placés les caractères à afficher, `x` et `y` définissant la position de ce texte à l'écran.

b. Définir une fenêtre composée d'une zone de dessin et d'un bouton `Quitter`.

c. L'affichage de l'histogramme étant réalisé dans la zone de dessin,

- Le constructeur de la fenêtre doit prendre en paramètre un objet de type `Stat` de façon à le transmettre au constructeur de la zone de dessin.
- La méthode `paint()` définie dans la classe représentant la zone de dessin fait appel à la méthode `s.dessine()`, où `s` est un objet de type `Stat`, initialisé dans le constructeur de la zone de dessin.

d. Le bouton `Quitter` et l'icône de fermeture située dans la barre de titre de la fenêtre ayant la même fonctionnalité (quitter l'application et fermer la fenêtre) :

- Créer une classe `GestionQuitter` qui implémente l'écouteur `ActionListener` et dérive de la classe `WindowAdapter`.
- Définir les méthodes correspondant au comportement de fermeture d'application.

La saisie d'un numéro de compte

La classe `Saisie` décrite ci-dessous permet la saisie d'une chaîne de caractères par l'intermédiaire d'une fenêtre de saisie :

```
import java.awt.*;
import java.awt.event.*;
public class Saisie implements ActionListener {
    TextField réponse;
    public Saisie ()    {
        Frame F = new Frame ("Saisie de valeurs:");
        F.setSize(300, 50);
        F.setBackground(Color.white);
        F.setLayout(new BorderLayout());
        F.add (new Label("Valeur :"), "West");
        réponse = new TextField(10);
        F.add(réponse, "East");
        réponse.addActionListener(this);
        F.show();
    }
    public void actionPerformed(ActionEvent evt) {
        String numéro = réponse.getText();
        System.out.println(numéro);£££ accent ?£££
    }
}
```

Observer et analyser cette classe et transformer la méthode `actionPerformed()` de façon à calculer puis à afficher l'histogramme cumulé si le numéro de compte lu dans la fenêtre de saisie correspond à un compte enregistré dans le fichier `Compte.dat`.

Contenu et exploitation du CD-Rom

Le CD-Rom fourni avec cet ouvrage est composé de :

- Deux fichiers au format PDF
  - outils.PDF ;
  - corriges.PDF.
- Quatre dossiers (répertoires) :
  - Java2 ;
  - Winzip ;
  - Acrobat ;
  - Sources.
- Un fichier `Lire.java`

Le fichier outils.PDF

ATTENTION ! Ce fichier est à lire avant d'installer le JDK et de compiler votre premier programme Java.

Ce fichier, au format PDF (*à lire avec le logiciel Acrobat Reader*), décrit :

- comment installer le JDK,
- comment construire son propre environnement de travail.

Vous y trouverez également des adresses Internet depuis lesquelles vous pourrez télécharger des environnements de développement pour MacOS, Unix ou Windows.

Le fichier corriges.PDF

Ce fichier, au format PDF (*à lire avec le logiciel Acrobat Reader*), présente, pour chaque chapitre du livre :

- le résumé,
- les exemples,
- le corrigé **commenté** et **expliqué** des exercices,
- le corrigé **commenté** et **expliqué** du projet.

Le dossier Java2

Dans ce dossier sont placés les programmes d'installation du JDK et sa documentation au format compressé.

- Pour installer le JDK, lisez le fichier `outils.PDF`.
- Pour décompresser la documentation, utilisez l'utilitaire Winzip.

Le dossier Winzip

Dans ce dossier, se trouve l'outil d'installation de l'application Winzip.

- Pour installer le logiciel et pour décompresser la documentation Java, lisez le fichier `outils.PDF`.

Le dossier Acrobat

Ce dossier contient l'outil d'installation de l'application Acrobat Reader . Ce logiciel vous permet de lire les fichiers au format .PDF, par exemple, le fichier `corriges.PDF` qui donne les corrigés des exercices et du projet.

- Pour installer cette application, double-cliquez sur l'icône d'installation située dans le dossier `Acrobat` et validez les requêtes du programme d'installation.

Le dossier Sources

Le dossier `Source` se compose de trois sous-répertoires : `exemples`, `exercices` et `projet`. Ceux-ci contiennent respectivement douze sous-répertoires, un pour chacun des chapitres :

```
introduction, chapitre1, chapitre2, chapitre3, …, chapitre11.
```

Chacun de ces répertoires contient les fichiers source des programmes correspondant :

- aux exemples. Ainsi, pour retrouver les programmes donnés en exemple au chapitre 1, allez dans le répertoire `sources/exemples/chapitre1`.
- aux exercices corrigés. Ainsi, pour retrouver le programme de l'exercice 2 du chapitre 4, allez dans le répertoire `sources/exercices/chapitre4`.
- au projet. Ainsi, pour retrouver le corrigé du projet du chapitre 5, allez dans le répertoire `sources/projet/chapitre5`.

Le fichier Lire.java

Très utilisé tout au long de l'ouvrage, ce fichier est à copier sur votre disque dur. Pour plus de précision, reportez-vous à la rubrique « Construire son propre environnement de travail » du fichier `outils.PDF`.

Index

A

B

C